Static Code Analysis Tools for Identifying High-Risk Software Modules: An Overview

Prepared for

NASA Independent Verification and Validation Facility

FAU Technical Report TR-CSE-00-21

Taghi M. Khoshgoftaar*
Edward B. Allen
Florida Atlantic University
Boca Raton, Florida USA

July 2000

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.

2 What is "measurement"?

2.1 Foundations

Measurement theory provides a body of results that can guide the formal definition of software metrics. Fenton and Zuse each give an overview of the representational theory of measurement as applied to software measurement [26, 27, 118]. Krantz et al., and Roberts each present theoretical foundations [77, 103]. This section presents some results from measurement theory. Consider the following basic definitions.

An *empirical relation system*, $\mathcal{C} = (C, R)$, consists of a set of objects, C, and a set of relations on them, R [26].

A *numerical relation system*, $\mathcal{N} = (\Re, P)$, consists of a number system, and a set of relations, P. The real number system, $\Re$, is preferred for software measurement, due to its rich set of relations that are familiar to engineers.

A *measure* is a mapping from an empirical relation system into a numerical relation system.

$$M : C \rightarrow R \tag{1}$$

It must map each object to a number, and each empirical relation to a numerical relation, such that the mapping is a homomorphism, i.e., such that all relations in the empirical relation system hold if and only if the corresponding numerical relations hold. For example, temperature is a measure of an attribute of physical objects. The empirical relation system embodies our common-sense ideas about heat. The numerical system is real numbers. If temperature numbers behave in a way that matches our common-sense empirical system, then it can be shown to be an appropriate measure. A *software metric*

is a measure of an attribute of a software quality, resource, product, process, or execution.

Formal definition of new software metrics consists of the following stages [29].

1. Identify an attribute for some real-world entities.
2. Identify empirical relations for the attribute.
3. Identify numerical relations corresponding to each empirical relation.
4. Define a mapping from real-world entities to numbers.
5. Check that numerical relations preserve and are preserved by empirical relations.

2.2 Scale types

A *scale* is defined as a triple consisting of an empirical relation system, a numerical relation system, and a measure, $(\mathcal{C}, \mathcal{N}, M)$. Scales may be categorized according to their properties. As listed in Table 1, the following are classic *scale types*.

- *Nominal* scale is essentially categorization.
- *Ordinal* scale is characterized by a strict weak order among objects.
- *Interval* scale adds the concept of distance to ordinal scale properties.
- *Ratio* scale adds the concept of zero measurement to the properties of interval scales.
- *Absolute* scale is essentially counting.

Table 1: Classic scale types

Type	Key Concept (Cumulative)
Nominal	Classification
Ordinal	Strict weak order
Interval	Distance
Ratio	Zero
Absolute	Counting

Table 2: Admissible transformations

Scale Type	Transformation	Example $\phi(x)$
Nominal	one to one	(relabel items)
Ordinal	monotonic increasing	$x \geq y \Leftrightarrow \phi(x) \geq \phi(y)$
Interval	positive linear	$\phi(x) = \alpha x + \beta, \alpha > 0$
Ratio	multiply by positive constant	$\phi(x) = \alpha x, \alpha > 0$
Absolute	identity	$\phi(x) = (1)x$

For example, degrees Fahrenheit and Celsius are interval scales of temperature. Degrees Kelvin is a ratio scale of temperature because absolute zero is part of the empirical system of chemistry. Other scale types could be devised, such as a scale type for probability.

Each classic scale type is distinguished by the class of transformations that preserves the underlying empirical relations. A statement involving numerical scales is *meaningful* if its truth remains unchanged if every scale involved is replaced by an admissible transformation, as listed in Table 2, where ϕ is a transformation, x and y are measurements, and α and β are constants.

A transformation is usually a conversion from one unit of measure to another. For example, conversion from Fahrenheit, F, to Celsius, C, is done by $C = (F + 32)(5/9)$, a

positive linear transformation of a measurement, which is appropriate for interval scales.

Nominal scale is considered the "weakest" and absolute scale is considered the "strongest". Note that a strong transformation is also a member of all the weaker classes. When one applies an inadmissible transformation to a measurement, one is implicitly assuming additional empirical relations.

Certain common statistical calculations have been categorized by class of admissible transformations [97]. For example, the following are admissible ways to calculate the "center" of a set of samples.

- *Nominal* scale: Mode
- *Ordinal* scale: Median
- *Interval* scale: Arithmetic Mean
- *Ratio* scale: Geometric Mean

A *meaningful* statement about a measurement(s) is one where the empirical truth of a statement about measurements does not depend on the unit of measure. Table 3 lists statistics and the classic scale type corresponding to each.

Roberts [103] presents criteria for scale types. Ordinal scale is useful for comparing and for prioritizing a set of modules. The median is the appropriate measure of the "center" of a sample of measurements. Interval scale allows more sophisticated statistical analysis than ordinal scale. For example, the arithmetic mean is the appropriate measure of the "center" of a sample of measurements. Ratio scale is the most familiar in daily life. It allows most common analysis techniques.

Table 3: Meaningful statistics

Scale	Relations	Statistics	Test
Nominal	Equivalence	Mode Frequency	Nonparametric
Ordinal	Greater than	Median Spearman corr.	Nonparametric
Interval	Ratio of intervals	Mean Std Dev Pearson corr	Parametric
Ratio	Ratio of values	Geometric mean Coeff of Var	Parametric

Transformations of measures pertains to individual measures. We are also interested in formal relationships among multiple measures, and in particular, where one synthetic measure is derived from combining primitive measures, or a dependent variable is derived from independent variables in a predictive mathematical model. According to Roberts [103], there is no generally accepted theory regarding a scale derived from primitive measures. However, one can define a derived scale in a narrow sense or a wide sense [103, pp.76–80][26]. See the references for further details.

2.3 Controversy over scale types

There has been considerable controversy during the 1990's in the software metrics community, over whether or not scale types of software metrics should limit one's choice of analysis and modeling methodologies. From even earlier, this controversy has been ongoing in the statistics community. "Statisticians have generally rejected the proscription of methods based on the limitations of permissible transformations." Proscribing statistics based on scale type does not work [111].

The viewpoint of Mathematics is different from the viewpoint of Science. Axiomatic measurement theory, as advocated by some software-metrics researchers, is mathematics. Science, as advocated by other software-metrics researchers, discovers new empirical relations, which may not be "meaningful" in preliminary work, due to the lack of substantive theory. One should keep in mind that scale type is defined in the context of empirical relation systems.

Measurement theory is still developing. For example, measurement theory dealing with statistical error is not well developed. We recommend a pragmatic approach to measurement theory as advocated by Briand, El Emam, and Morasca [14]. We suggest that a software-metrics analyst ignore scale type when doing exploratory statistics. However, one should use scale type to evaluate conclusions. Measurement theory can help one evaluate whether conclusions are nonsense [111].

3 What should we measure?

A measurement *subject* is the object under study, such as a body of software and related artifacts. We are interested in measuring attributes that are relevant to the software development.[2] In particular, measurement subjects include the following.

- Qualities of the software in the context of operations or during maintenance
- Resources used in development
- Products that result from development
- Processes that occur during development

[2]In this context, we consider maintenance as a type of development.

- Execution of the software

3.1 Quality

"Beauty is in the eye of the beholder" indicates the inherent difficulty in measuring software quality. From a practical viewpoint, there are various *quality* attributes each in relation to a different aspect of software development. "This software is *good* for such-and-such." These attributes are called *quality factors* [105], such as functionality, reliability, efficiency, usability, maintainability, and portability.

Our research has focused on indicators of reliability. *Reliability* is usually defined as the probability of failure-free execution. A *failure* is incorrect execution, usually in the context of operations. According to standard terminology, a *fault* is a defect in a program that may cause incorrect execution [81]. Faults are caused by errors committed by people. Many of our studies focus on the absence of faults as recorded by a problem reporting system during the period of interest. This is an indicator of reliability, in a broad sense, irrespective of whether a failure resulting from a fault would be frequent or rare during operations. Even rare failures can be very important in mission-critical systems.

3.2 Resources

Software development is a human-intensive activity. The most significant resource needed to produce software is the *efforts* of appropriately skilled people. Many researchers and managers have sought ways to measure the *productivity* of software developers [29]. This has been very difficult to do in a meaningful way, because the same product is not

produced over and over, and because the combination of many skills are needed for success.

Because of the economic context of most software development, managers must predict the amount and kinds of effort that will be needed to develop a prospective software product. Effort estimation has also been very difficult to do with the accuracy desired by business leaders [29].

3.3 Products

Software *product metrics* are quantitative attributes of abstractions of software. Software can be viewed as a *static* product embodied by source code, or it can be viewed as a *dynamic* product while it is executing. This section focuses on static attributes of software artifacts. However, one should bear in mind that the dynamic behavior of software can be statically portrayed by diagrams which are subject to static measurement.

Commonly measured static product abstractions include call graphs [89], control flow graphs, statements, and object-oriented programming abstractions [10, 19]. (Some of these abstractions may be created during the design phase or they may be extracted from code in a later phase [1, 6, 23, 117].) For example fan-in and fan-out [37, 91, 116] are attributes of a node in a call graph, where each node is an abstraction of a module and each edge represents a call from one to another. Many software product metrics are attributes of a control flow graph in which the nodes represent decision statements or branch destinations and the edges represent potential flow of control, similar to a flowchart. McCabe's cyclomatic complexity is one of the best known in this category [83, 84]. Lines-of-code is the best known statement metric. Other examples are Halstead's

counts of operators and operands [33].

Properties of attributes. Briand, Morasca, and Basili, propose a set of properties in a formal framework to define measures of common software attributes more precisely [16]. They define the following families of measures on graphs that have certain properties in common.

- Size
- Length
- Complexity
- Coupling
- Cohesion

One could similarly define other families of metrics for attributes of interest to software engineers. More recently, Morasca and Briand extended their framework from graphs (binary relations) to relations in general [85]. We adopt their definitions, which can be used to define measures of many kinds of software design abstractions, and thus, many kinds of software metrics.

Briand, Morasca, and Basili [16] define a *modular system* as the context for their measures, where a module is a subsystem.

Definition 1 (Modular System [16])

A modular system, *MS, is an abstraction of a software system represented by a graph with* n *nodes partitioned into modules,* $m_k, k = 1, \ldots, n_M$.

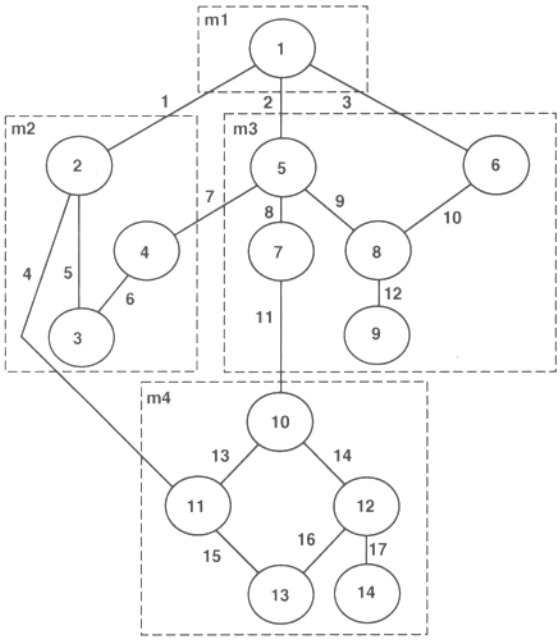

Figure 1: Example of a modular system [2]

Figure 1 is an example of a modular-system graph. This is similar to Figure 2 in [16]. The modules (dashed boxes) partition the nodes in the system.

Briand, Morasca, and Basili [16] propose a set of properties that defines the concept of the *size of a system*. Table 4 summarizes the properties of this family of measures, and corollaries derived from the properties. In a software metrics context, we reserve the term *size* for measures that have these properties. In addition, corresponding properties of the size of a module, shown in Table 5, can be derived from the system-level properties in Table 4.

Briand, Morasca, and Basili [16] also propose a set of properties that defines the concept of the *length of a system*. Table 6 summarizes the properties of this family of measures. In a software metrics context, we reserve the term *length* for measures that have these properties.

Table 4: Properties of the size of a system [16]

1. Nonnegativity. The size of a system is nonnegative.
2. Null value. The size of a system is null if its set of nodes is empty.
3. Module additivity. Given a system, $\mathbf{S}$, having modules, m_1 and m_2, such that every node in $\mathbf{S}$ is in m_1 or m_2, but not both, the size of this system is equal to the sum of the sizes of the modules m_1 and m_2.

$$Size(\mathbf{S}) = Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

Corollaries

1. Node additivity. Given a modular system, MS, where each node is a module, $m_i, i = 1, \ldots, n$, the size of the modular system is given by

$$Size(MS) = \sum_{i=1}^{n} Size(m_i|MS)$$

2. Monotonicity. Adding a node to a system does not decrease its size.
3. General module additivity. Given a system, $\mathbf{S}$, with any two modules, m_1 and m_2 such that every node in $\mathbf{S}$ is a node in m_1 or m_2 or both, the size of the system is not greater than the sum of the sizes of the pair of modules.

$$Size(\mathbf{S}) \leq Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

4. Merging of modules. Given a system, $\mathbf{S}$, with any two modules, m_1 and m_2 such that every node in $\mathbf{S}$ is a node in m_1 or m_2 or both, construct $\mathbf{S}'$ such that m_1 and m_2 in $\mathbf{S}$ are replaced by $m_{1\cup 2} = m_1 \cup m_2$ in $\mathbf{S}'$.

$$Size(\mathbf{S}') \leq Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

Table 5: Properties of the size of a module [16]

1. Nonnegativity. The size of a module is nonnegative.
2. Null value. The size of a module is null if its set of nodes is empty.
3. Module additivity. Given a module, m_k, in a system, $\mathbf{S}$, having modules within it, m_1 and m_2, such that every node in m_k is in m_1 or m_2, but not both, the size of this module is equal to the sum of the sizes of the modules m_1 and m_2.

$$Size(m_k|\mathbf{S}) = Size(m_1|\mathbf{S}) + Size(m_2|\mathbf{S})$$

Table 6: Properties of the length of a system [16]

1. Nonnegativity. The length of a system is nonnegative.
2. Null value. The length of a system is null if its set of nodes is empty.
3. Nonincreasing monotonicity for connected components. Given a system, $\mathbf{S}$, with a module, m_k, consisting of a connected component, adding an edge to m_k does not increase the length of $\mathbf{S}$.
4. Nondecreasing monotonicity for non-connected components. Given a system, $\mathbf{S}$, with two modules, m_1 and m_2, each consisting of a distinct connected component, adding an edge between a node in m_1 and a node in m_2 does not decrease the length of $\mathbf{S}$.
5. Disjoint modules. Given a system, $\mathbf{S}$, composed of two disjoint modules m_1 and m_2, the length of this system is the maximum of the lengths of the modules.

$$Length(\mathbf{S}) = \max(Length(m_1|\mathbf{S}), Length(m_2|\mathbf{S}))$$

Table 7: Properties of the complexity of a system [16]

1. Nonnegativity. The complexity of a system is nonnegative.
2. Null value. The complexity of a system is null if its set of edges is empty.
3. Symmetry. The complexity of a system does not depend on the convention chosen to represent the direction of edges.
4. Module monotonicity. Given a system, $\mathbf{S}$, with any two modules, m_1 and m_2, that have no edges in common, the complexity of the system is no less than the sum of the complexities of the two modules.

$$Complexity(\mathbf{S}) \geq Complexity(m_1|\mathbf{S}) + Complexity(m_2|\mathbf{S})$$

5. Disjoint module additivity. Given a system, $\mathbf{S}$, composed of two disjoint modules, m_1 and m_2, the complexity of the system is equal to the sum of the complexities of the two modules.

$$Complexity(\mathbf{S}) = Complexity(m_1|\mathbf{S}) + Complexity(m_2|\mathbf{S})$$

Corollary

- Monotonicity. Adding an edge to a system does not decrease its complexity.

Briand, Morasca, and Basili [16] further propose a set of properties that defines the concept of the *complexity of a system*. Table 7 summarizes the properties of this family of measures. In a software metrics context, we reserve the term *complexity* for measures that have these properties.

Briand, Morasca, and Basili [16] also propose a set of properties that defines the concept the *coupling of a modular system*, where a *module* is a subsystem. Table 8 summarizes properties of coupling of graphs. We reserve the term *coupling of a modular system* for measures that have the properties in Table 8, which are essentially the same as

Table 8: Properties of coupling of a modular system [16]

Concept/Properties
1. Nonnegativity. Coupling of a modular system is nonnegative.
2. Null value. Coupling of a modular system is null if its set of intermodule edges is empty.
3. Monotonicity. Adding an intermodule edge to a modular system does not decrease its coupling.
4. Merging of modules. If two modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the coupling of the modular system with $m_{1\cup 2}$ is not greater than the coupling of the modular system with m_1 and m_2.
5. Disjoint module additivity. If two modules, m_1 and m_2, which have no intermodule edges between nodes in m_1 and nodes in m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the coupling of the modular system with $m_{1\cup 2}$ is equal to the coupling of the modular system with m_1 and m_2.

in [16]. We reserve the term *coupling of a module* for measures that have the properties in Table 9, which are essentially the same as in [16]. However, we substitute "intermodule" for "output" edge in Properties 2 and 3. Practically speaking, our only difference with Briand, Morasca, and Basili, is we make no distinction between inbound and outbound coupling of a module.

Lastly, Briand, Morasca, and Basili [16] propose a set of properties that defines the concept *cohesion of a modular system*. Table 10 summarizes the properties of this family of measures and a corollary. Briand, Morasca, and Basili [16] also propose the set of properties shown in Table 11 that defines the concept *cohesion of a module* where a *module* is a subsystem.

Table 9: Properties of coupling of a module [16]

Concept/Properties
1. Nonnegativity. Coupling of a module is nonnegative.
2. Null value. Coupling of a module is null if its set of intermodule edges is empty.
3. Monotonicity. Adding an intermodule edge to a module does not decrease its module coupling.
4. Merging of modules. If two modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is not greater than the sum of the module couplings of m_1 and m_2.
5. Disjoint module additivity. If two modules, m_1 and m_2, which have no intermodule edges between nodes in m_1 and nodes in m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the module coupling of $m_{1\cup 2}$ is equal to the sum of the module couplings of m_1 and m_2.

Table 10: Properties of cohesion of a modular system [16]

Concept/Properties
1. Nonnegativity and Normalization. Cohesion of a modular system belongs to a specified interval, $Cohesion(MS) \in [0, \text{Max}]$.
2. Null value. Cohesion of a modular system is null if its set of intramodule edges is empty.
3. Monotonicity. Adding an intramodule edge to a modular system does not decrease its cohesion.
4. Merging of modules. If two unrelated modules, m_1 and m_2, are merged to form a new module, $m_{1\cup 2}$, that replaces m_1 and m_2, then the cohesion of the modular system with $m_{1\cup 2}$ is not greater than the cohesion of the modular system with m_1 and m_2.

operations. *USAGE* is the fraction of installations that had the given module installed; it can be forecast from deployment records [43]. A case study of a telecommunications system collected execution time measurements [62], *RESCPU*, *BUSCPU*, and *TANCPU*, which can be derived from laboratory measurements of an earlier release using simulated workloads on a standard hardware configuration. Each metric represents the average execution time in the given module of a transaction in the specified workload. Other systems could measure execution time in a similar manner. Future research will refine these metrics.

A case study for the EMERALD team [43] focused on customer-discovered faults, rather than faults discovered during testing. The likelihood of failure during a period of operations depends on both the likelihood of the existence of faults and the likelihood of execution of code. Faults are discovered by customers only when the faulty code is executed. Operational profiles are a tool of software reliability engineering to relate faults and failures during operations [90]. When test inputs conform to an operational profile, one can measure an execution profile of the system. An execution profile of a software system consists of the probability of execution of each module. Consequently, an execution profile can be viewed as a model of the exposure faulty code gets during operations. Our working hypothesis is the intuitive notion that more exposure implies more faults may be discovered by customers.

However, empirical data to calculate an execution profile may be difficult to obtain. Our study showed that deployment records of past releases can be a valuable source of data for calculating an approximation to the probability of execution.

Empirical data collection to support an execution profile can have many practical problems. Data covering an entire customer-base is often difficult to obtain. Access to

user sites may be limited when they belong to customer companies. A large number of user sites in remote locations may multiply data collection costs. Execution profiles of some systems are technically challenging to measure directly in an operational setting. For example, embedded, real-time computer systems often do not lend themselves to instrumentation outside the laboratory.

We propose a new module-level metric, *USAGE*, defined as the proportion of customer systems with the module deployed. *USAGE* is an approximation of an execution profile under the assumption that if a module is deployed, it is also used. In contrast, an ideal execution profile measures the probability of execution directly. Our measure of *USAGE* currently has a range from zero to one, but future measures may be scaled differently.

When the developer's organization is involved in deployment at each customer site, records of that activity are accessible. In a stable market, *USAGE* can be derived from past deployment data, perhaps modified by current plans, without further intrusion at customer sites. Telecommunications software is a good example of this class of software.

We analyzed a metric derived from deployment records which is a practical surrogate for an execution profile in the context of a software quality model of a legacy system. We define "usage" as the proportion of systems in the field which have a module deployed. A case study of a very large telecommunications system investigated the significance of usage in the context of a software quality model and found it was very useful [43].

USAGE is currently incorporated into EMERALD models. The EMERALD team anticipates that refinements will be as well. The positive results of their study [43] demonstrated the importance of execution profiles. Consequently, they are currently investing in collection of execution profile data which will be a closer approximation to an ideal

execution profile.

4 How should we measure?

A *measurement protocol* is a set of conditions that assures consistent repeatable measurement of an attribute [75]. A valid protocol is independent of the measurer and the measurement environment. A valid protocol is also compatible with the desired unit of measure and the purpose of the measurement. Given a software system, various protocols derive abstractions that are suitable for measurement.

4.1 Aggregation

Software entities can be measured at various levels of aggregation, ranging from an entire system to individual tokens. We use the word *module* for the smallest entity that is measured in a study; the size varies considerably from study to study. The following are common levels of aggregation for software measurement, from largest to smallest.

- System
- Subsystem (on various levels)
- Abstract data type
- Source file
- Object
- Function (method)

Measurements of small entities can sometimes be arithmetically combined as a measurement of a higher-level entity.

4.2 Abstractions

The approach of Briand, Morasca, and Basili [16] is compatible with any measurement protocol that produces a graph representing some aspect of software design. Many design methods produce graphs as artifacts. An artifact created during the design phase represents the design decisions made at that time, and embodies the abstraction of the software that was used by the designer.

The procedural development paradigm typically produces certain kinds of artifacts, while the object-oriented paradigm produces other kinds of artifacts. The following are examples of different abstractions of software. Attributes of each abstraction could be subject to measurement.

- Architecture graphs
- Formal/informal model diagrams, such as UML or SDL diagrams.
- Data-flow diagrams
- Call graphs
- Inheritance diagrams
- Message passing diagrams
- Set/use graphs for global variables
- Control-flow graphs

- Statements
- Tokens

Measurements of these abstractions will be valuable to the extent they reveal something about the intellectual process of developing software.

Briand, Daly, and Wüst [12, 13] survey numerous proposed measures of coupling and cohesion for object-oriented designs. Various abstractions are measured, such as class inheritance, class type, method invocation, and class-attribute references, and thus, various software attributes are measured.

Bieman et al. propose cohesion measures based on input/output dependence graphs of designs [6], and slicing of code [7].

5 What is the standard for a measurement?

A *measurement instrument* is a tool for mapping an abstraction of software to a number or category [75]. An instrument is usually designed to detect a particular unit of measure, and assures that each unit of an attribute is equivalent within the constraints of measurement error. A measurement instrument provides a *standard* unit of measurement. Most traditional software metrics count artifact features, as though each item is equal in the mind of a designer. The "measurement instrument" in each case is an algorithm for detecting a feature, plus counting.

5.1 Units of measure

For example, Briand, Daly, and Wüst [13] propose an integrated measurement framework for object-oriented coupling metrics, based on counts. The framework successfully encompasses many metrics proposed in the literature. Each qualifying metric is equivalent to the number of edges of an appropriate graph. The unit of measure is the meaning of an edge.

They also propose a similar framework for object-oriented cohesion metrics [12]. Due to the properties of cohesion in Tables 10 and 11, cohesion is a ratio of like units where a unit of the numerator or denominator is the meaning of an edge.

5.2 Calculating a measurement

The method for calculating a software metric implies the unit of measure and is essentially the measurement instrument. This calculation is typically represented by an algorithm that processes an abstraction of the software entity.

6 Is a measurement theoretically valid?

Kitchenham, Pfleeger, and Fenton draw from measurement theory to propose criteria for theoretical validation of software metrics [75]. In their view, a theoretically valid measure of a software attribute should fulfill the following criteria.

1. Attribute validity criteria. Is the attribute exhibited by the entity of interest?

   (a) Distinguishing entities. A valid measure must distinguish different entities from one another.

(b) Representation condition. A valid measure must preserve our intuitive notions about the attribute.

(c) Equivalent units. Each unit of an attribute is equivalent within the constraints of measurement error.

(d) Same value allowed. Different entities may have the same measured attribute value within the limits of measurement error.

2. Unit validity criteria. Is the unit of measure appropriate? The measurement scale-type should fit its intended use. An alternative unit for the same attribute should be an acceptable transformation of an established unit of measure.

3. Instrument validity criteria. Does the instrument specify how to capture measurement data? A valid instrument should be accurate in a given unit of measure. In particular, the instrument implements the criterion of Equivalent Units above.

4. Protocol validity criteria. Does the protocol assure consistent, repeatable measurements that are independent of the measurer and the measurement environment? A valid protocol should be unambiguous, self-consistent, and accepted by the software engineering community.

Of course, these theoretical criteria do not address whether a measure is useful in software engineering practice [105]. Several studies have investigated the usefulness of various object-oriented coupling measures [3, 10, 11, 35].

7 Do measurements have practical value?

Scientists like to conduct controlled experiments to test theories about the laws of nature. Accordingly, many software-metrics researchers would like to discover the "laws of nature for software development". For example, one might investigate the question, "What are the causes of software errors?" Software quality models attempt to answer this by a statistical model of the relationship between software metrics and software errors. Unfortunately, due to the many human factors that influence software errors which cannot be measured, controlled experiments to evaluate the usefulness of software quality models are not practical [4, 98]. Therefore, we advocate the case-study approach as a useful alternative, even though it is incapable of proving "laws of nature".

7.1 Empirical studies

Case studies are practical demonstrations of the usefulness of software quality models [105]. In other words, a case study addresses the question, "Does a software quality model have useful statistical accuracy and robustness to predict the value of a measure of software errors?" A case study evaluates accuracy and robustness in a limited real-world context.

Each scale-type for a dependent variable demands a different kind of empirical modeling technique. A nominal-scale dependent variable means a classification technique should be used. An ordinal-scale dependent variable requires a rule for ordering modules [48]. A ratio-scale dependent variable that is restricted to integers, such as the number of errors, calls for a modeling technique such as Poisson regression. If a ratio-scale dependent variable is modeled as a real number, such as the size of changes, then a wide variety

of linear and nonlinear regression techniques may be applicable [66, 70], beginning with multiple linear regression.

In a typical case study, we access historical data on one or more past projects where actual software errors are known. A problem reporting system captures this information. Software-attribute measures are considered independent variables. Configuration management systems support recovery of archived versions of software artifacts. A case study constructs models that could have been developed during the historical project, and calculates predictions that could have been made. The accuracy of those predictions is then evaluated against the actual errors. A typical empirical case study will include the following steps [65].

1. Retrieve a baseline artifacts of a historical project's software development from the configuration management system, e.g., source code.

2. Measure errors in modules from the historical project, based on data in a problem reporting system.

3. Measure module attributes and attributes of the model-building process.

4. Prepare historical data as a *fit* data set and a *test* data set. The test data set should be independent of the fit data set. When only one project's data is available at a time, we can randomly split the data into fit and test data sets.

5. Construct an empirical model by selecting significant independent variables and estimating parameters, using an appropriate empirical modeling technique with the fit data set.

6. Evaluate the empirical model's accuracy by predicting a measure of errors in each module in the test data set, and comparing the predictions to the actual value. We evaluate accuracy using statistics appropriate to the scale-type of the dependent variable.

In the statistics literature, regression modeling studies typically use the terms *fit* and *test* data sets, as explained above. However, in the classification and machine-learning literature, the same data sets are often called *training* and *evaluation* data sets, respectively. Moreover, in the regression literature, variables are categorized as a *dependent* variable or *independent* variables, but the classification and machine-learning literature speaks of a *response* variable and *predictors*. This difference in terminology is merely an artifact of history. In this paper, we use whichever terminology is traditional for the context.

7.2 Threats to validity

Threats to internal validity are unaccounted influences that may affect case study results. In software engineering practice, software faults are caused by a wide variety of conditions. The number of faults attributed to each module may have been influenced by things that were not measured. We mitigate this threat to internal validity by defining measures in many dimensions on various abstractions of the design.

Threats to external validity are conditions that limit generalization of results. We often use the case study approach, because controlled experiments are often performed in artificial settings that are not realistic enough to be externally valid. Case studies also have inherent limits on the generalizability of results due to the wide variety of software

development organizations. To be credible, the software engineering community demands that the subject of an empirical study be a system with the following characteristics [114].

1. Developed by a group, rather than an individual;
2. Developed by professionals, rather than students;
3. Developed in an industrial environment, rather than an artificial setting;
4. Large enough to be comparable to real industry projects.

We fulfill all of these criteria through collaborative arrangements with software development organizations, and thus, avoid threats-to-validity due to artificial settings.

Multiple case studies provide a sampling of possible dependent variables, and evidence of consistency of results across software projects, because software applications have distinct product characteristics, and software is developed under a variety of conditions in various organizations.

8 How do we use measurements?

Application of a software quality model is when measurements on a system currently under development are input to the model, so that predictions can be calculated. At that point in time, one does not know whether the predictions are true, but one can make management and design decisions based on the predictions, because a relevant case-study achieved a certain level of accuracy.

For example, EMERALD provides access to metrics of source code, problems related to source code, and usage profiles [39]. It also provides access to risk models based on

those metrics. For example, at the time of a high-level design review for a new release, suppose EMERALD indicates that certain modules are at risk when changed. After a review, the design team may choose to reengineer these modules first, before continuing implementation of the new release. Risk assessments can also be valuable when assigning technical staff to test, test automation, and maintenance efforts, and when matching skill level and experience with complexity. Moreover, during maintenance, any changes proposed for high risk modules can entail more stringent justification and inspection.

9 Modeling details

A recent status report [100] on the field of software measurement highlights gaps between current research and practice. For example, practitioners want accurate, timely predictions of which modules have high risk, but researchers have yet to find adequate, widely applicable measures and models. Faults are a result of mistakes or omissions by developers, and relevant human behavior in the workplace is notoriously difficult to measure directly. In other words, there is no comprehensive scientific theory linking software metrics and software errors [28].

We take a more pragmatic approach as described in [60] and in [53]. We capture relevant variation among modules with practical metrics, even though the underlying human behavior is not well understood. Instead of expensive, specialized data collection, we leverage existing databases collected for other purposes, so that the marginal cost of data collection is modest. Rather than waiting for researchers to formulate a general theory, we achieve useful accuracy by empirically calibrating models to each local development environment.

Fayyad [24] defines *knowledge discovery in databases* as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data." Fayyad restricts the term *data mining* to one step in the knowledge-discovery process, namely, extracting patterns or fitting models from data. Others use the term more broadly. "Primary data analysis" in statistics is motivated by a particular set of questions that are formulated before acquiring the data. In contrast, data mining analyzes data that has been collected for some other reason. Hand [34] defines data mining as "the process of secondary analysis of large databases aimed at finding unsuspected relationships which are of interest or value to the database owners." Data mining is most appropriate when one seeks valuable bits of knowledge in large amounts of data collected for some other purpose, and when the amount of data is so large that manual analysis is not possible.

This aptly describes software quality modeling, especially when operational faults are rare. Many software development organizations have very large databases for project management, configuration management, and problem reporting which capture data on individual events during development. For large legacy systems or product lines, the amount of available data can be overwhelming. Manual analysis is certainly not possible. However, we have found that these databases do contain indicators of which modules will likely have operational faults.

Given a set of large databases or a data warehouse, Fayyad et al. give a framework of major steps in the knowledge-discovery process [25]: (1) selection and sampling of data; (2) preprocessing and cleaning of data; (3) data reduction and transformation; (4) data mining; and (5) evaluation of knowledge. We apply Fayyad's framework to predicting software quality from software development databases. We extracted knowledge from a

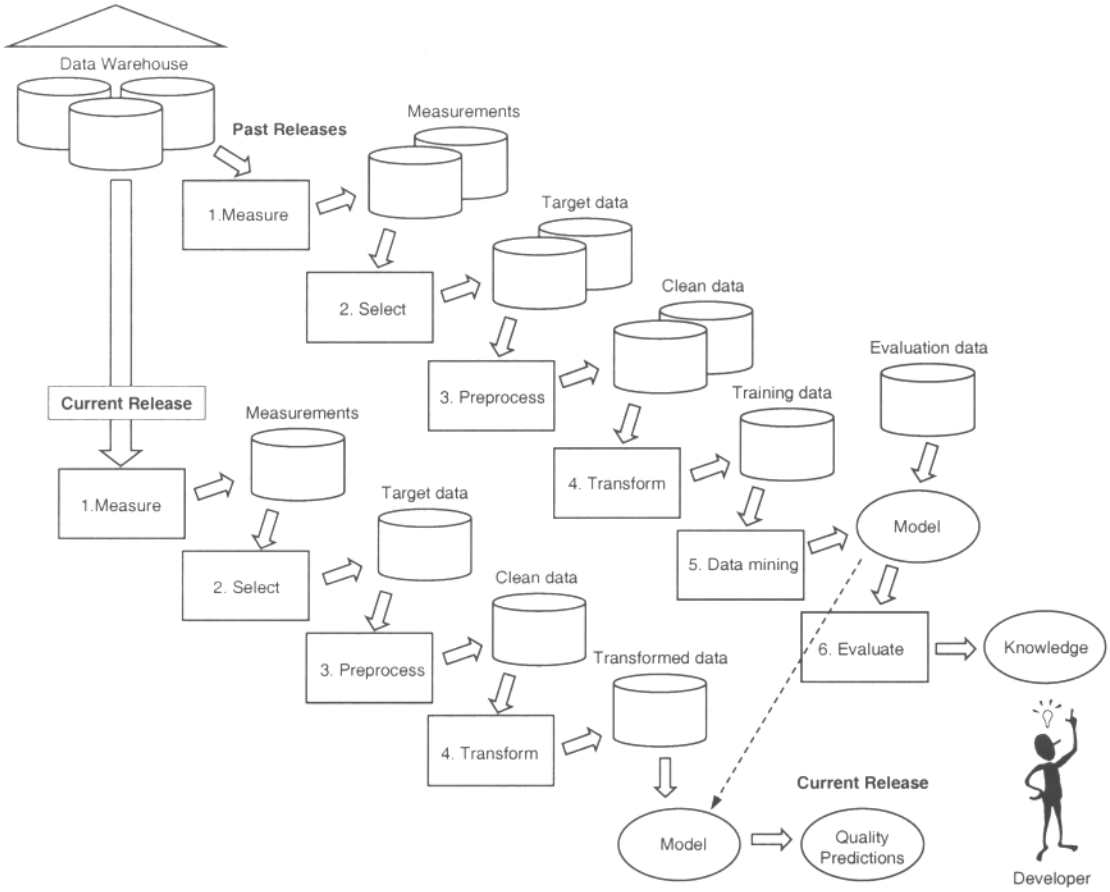

Figure 2: Exploit your gold mines

very large legacy telecommunication systems's configuration management and problem reporting databases. Our framework is shown in Figure 2.

Figure 2 has two similar tracks of processing steps. The upper track processes data on past releases where fault data is known. The results of this track are an empirical model, an assessment of its accuracy, and an interpretation of its structure. The lower

track processes data on a current release that is still under development, predicting which modules will be fault-prone through the empirical model. The human figure in the corner represents a developer who will make use of the predictions, the expected accuracy, and the knowledge derived from the model's structure.

In Figure 2, the *Data Warehouse* represents software development databases, such as configuration management systems and problem reporting systems, irrespective of the storage system implementation.

In Figure 2 Step 2, Select, chooses data for study, resulting in target data. Step 3, Preprocess, accounts for missing data and outliers in the target data, resulting in clean data. Step 4, Transform, may extract features from the clean data, and may transform data for improved modeling. The result is separate transformed data sets for training and for evaluation. Step 5, Data mining, builds a model based on the training data. Step 6, Evaluate, assesses the model's accuracy using the evaluation data, and analyzes the model's structure.

9.1 Data collection

The first step in Figure 2 measures available software development databases to derive variables from source code, configuration management transactions, and problem reporting transactions for one or more past releases. Pragmatic considerations usually determine the set of available predictors. We do not advocate a particular set of software metrics for software quality models to the exclusion of others recommended in the literature. Because marginal data collection costs are modest, we prefer to apply data mining to a large well-rounded set of metrics rather than limit the collection of software

metrics according to predetermined research questions.

9.2 Data analysis

Univariate analysis. The following summarizes lessons learned in a study for the EMERALD team [59]. The data set produced by data collection had data on over 18 thousand modules. Over 15 thousand faults were attributed to the system under study.

Some modules records did not contain metric data. Investigation by the data collection team found that most of them were empty files. These were not considered in our analysis. Default values may need to be defined for missing metric values.

Because no metric in this study was meaningful for negative values, a negative minimum value indicated a data collection anomaly.

A standard deviation of zero means that the metric was constant for all modules. Such metrics are not useful as independent variables in a model.

A standard deviation greater than the mean indicates a distribution skewed toward the small end. Many metrics have a distribution with a sparse tail at the high end. Thus, one should avoid analysis techniques that assume a symmetric or even a "bell curve" distribution.

Because our long term goal is to develop models that can identify fault-prone modules and can predict faults, we consider the relationship between each metric and the total number of faults, rather than fault density (faults/size).

Individual metrics are moderately correlated to faults. Typically, the single metric with the highest correlation is lines of code, *LOC*. Other size related metrics have similar correlations. Recall that *correlation* is a linear relationship. However, because a majority

of modules have zero faults and most of the other modules have only a few, one would not expect a simple linear model to predict very well. More sophisticated multivariate models are often needed to achieve useful accuracy.

Practical considerations in using metrics also may be important, such as the following.

In this case study, a very large number of modules had no faults. Some of the modules were not in the metrics data set, and some faults were attributed to empty modules. These faults were not considered in the analysis. These anomalies represented very small fractions of the total modules and total faults. Therefore, we concluded that a data set omitting these anomalies is still generally representative of the system.

A significant number of modules had no edges in the control flow graph. Therefore, one should note that some modules may consist of declarations and/or data only.

We found that the number of independent paths, *PTHIND*, in a control flow graph may have such large values that a sum of squares overflows the usual arithmetic capacity of our statistical tool (greater than 10^{32}). This metric may be of practical use in comparison techniques such as decision trees, but is not suitable for common statistical techniques. One approach is to consider all modules with $PTHIND > 10,000$ to be equivalent for modeling purposes. Another approach is to transform the metric.

Principal components analysis. Considering the proliferation of software metrics that purport to measure "complexity", Munson and Khoshgoftaar demonstrated that such metrics, in fact, characterize multidimensional variability in software attributes [87]. Consequently, rather than focus on just one metric at a time, many studies have developed multivariate quality models. Khoshgoftaar, Szabo, and Woodcock [73] showed that a

multiple regression model can have better predictive quality than a simple regression model based on the best single metric. They reason that a model accounting for more sources of variability is likely to be more robust for a new project. Khoshgoftaar and Allen also illustrate that a multivariate classification model can be more accurate than a model based on the best single metric of the set [45].

All modeling techniques have limitations. Multivariate models can be misleading if the underlying metrics are highly correlated. Correlation among independent variables in a model is called *multicollinearity*. In such situations, insignificant variations in the data can result in drastically different model parameters. This condition is called an *unstable model*. Preliminary analysis in a case study [59] indicated that, with a few exceptions, most call graph metrics and control flow graph metrics are highly correlated with each other. This is indicative of multicollinearity.

Munson and Khoshgoftaar propose using principal components analysis to transform correlated metric data into orthogonal variables in order to avoid such problems [69, 87, 88]. Of course, there is no guarantee of improved results. After all, if the raw metrics are only moderately correlated, a raw metrics model may be satisfactory. In practice, one often finds that raw software metrics are highly correlated to each other, and thus, some safeguards are in order, such as principal components analysis.

Software product metrics have a variety of units of measure, which are not readily combined in a multivariate model. We transform all product metric variables, so that each standardized variable has a mean of zero and a variance of one. Thus, the common unit of measure becomes one standard deviation.

Principal components analysis is a statistical technique for transforming multivariate data into orthogonal variables, and for reducing the number of variables without losing

significant variation. The following summarizes our description of principal components analysis in [57]. Suppose we have m measurements on n modules. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where each row corresponds to a module and each column is a standardized variable. Our principal components are linear combinations of the m standardized random variables, $Z_1, \ldots, Z_m$. The principal components represent the same data in a new coordinate system, where the variability is maximized in each direction and the principal components are uncorrelated [107]. If the covariance matrix of $\mathbf{Z}$ is a real symmetric matrix with distinct roots, then one can calculate its eigenvalues, λ_j, and its eigenvectors, $\mathbf{e}_j, j = 1, \ldots, m$. Since the eigenvalues form a nonincreasing series, $\lambda_1 \geq \ldots \geq \lambda_m$, one can reduce the dimensionality of the data without significant loss of explained variance by considering only the first p components, $p \ll m$, according to some stopping rule, such as achieving a threshold of explained variance. For example, choose the minimum p such that $\sum_{j=1}^{p} \lambda_j / m \geq 0.90$ to achieve at least 90% of explained variance. Let $\mathbf{T}$ be the $m \times p$ standardized transformation matrix whose columns are defined as $\mathbf{t}_j = \mathbf{e}_j / \sqrt{\lambda_j}$ for $j = 1, \ldots, p$. Let D_j be a principal component random variable, and let $\mathbf{D}$ be an $n \times p$ matrix with D_j values for each column, $j = 1, \ldots, p$. $D_j = \mathbf{Z}\mathbf{t}_j$ and $\mathbf{D} = \mathbf{Z}\mathbf{T}$. When the underlying data is software metric data, we call each D_j a *domain metric.*

Transformations. We recommend that selection of product metrics be based on the following general principles [59]..

- Total counts are preferred over means and maximums for predicting the value of a quality factor.

- Counts of disjoint subsets are preferred over counts of overlapping subsets.
- Metrics which are linear combinations of others are not desirable because they cause singularities in some analysis techniques, and they do not contribute to multivariate models.
- Variables with extremely large values or values extremely close to zero are not practical to work with. A transformation is usually recommended.

Averages and maxima. Our models typically do not predict fault intensity, but rather are based on total faults. Intuitively, total software metrics are more closely related to this dependent variable. Total counts are usually ratio scale or stronger allowing straightforward synthesis of quality models. Means and maximums entail careful consideration of measure properties, because some models using them are not "meaningful" in the sense of measurement theory.

Overlapping subsets. Overlapping subsets are inherently correlated. Multicollinearity may confound underlying relationships with a quality factor, and thus, make interpretation of a model unnecessarily difficult. We should avoid using variables that we know by definition are significantly correlated.

Synthetic metrics. Sometimes the output of a metric analyzer includes quantities that represents a software quality modeling ideas. Many are based on a synthesis of primitive software metrics, rather than true software measurements. One should include synthetic metrics in a model only when one is purposely adopting the underlying modeling idea.

For example, EMERALD calculates the following synthetic metrics.

- Abductive Inference Machine. *AIM* is a composite complexity score.

- Test Targeting Risk. *TTRISK* is a composite complexity index made up of AIM and other factors.
- Complexity level. *LEVEL1*, *LEVEL2*, *LEVEL3*, *LEVEL4*, and *LEVEL5* each count the number of procedures at each level of the Datrix routine complexity model.
- Metrics out of range. *OUTRANGE* is the number of routines with metrics out of acceptable range, according to Datrix thresholds.
- Halstead metrics. *HALDIF*, *HALEFF*, *HALLEN*, *HALLVL*, *HALVOC*, and *HALVOL* are synthetic metrics defined by Halstead [33].
- Conditional arc complexity. *CNDCPLAV* and *CNDCPLMX* are metrics based on the Datrix model of conditional arc complexity.
- Volume of structures. *CTRVOL* and *LOPSTRAV* are metrics based on the Datrix model of control structure volume.
- Weighted metrics. *CTRBRCWT*, *ARCWT*, *CTRNSTWT*, and *LOPSTRWT* are metrics that weight arcs according to the number of statements in the arc.
- Statement complexity metrics. *STMCPLAV* and *STMCPLSM* are metrics based on the Datrix model of statement complexity.

Very large values. In general, we recommend a logarithmic transformation of variables that have a very large range (many orders of magnitude). We found in a study for the EMERALD team that the maximum value of the number of independent paths (*PTHIND*) was too large for computation by the SAS statistics package. A base 2 logarithmic transformation (*LGPATH*) was successfully employed by a case study [61].

In a study for the EMERALD team [61], their execution time metrics *RESCPU*, *BUSCPU*, and *TANCPU* had ranges of more than six orders of magnitude. The raw metrics were not significant in our logistic regression models based on product and process metrics. Among transformations of these variables, we found that only log *TANCPU* was significant, but the resulting model was not dramatically more accurate than the baseline model.

9.3 Prediction

Classification. A classification model has a dependent variable that has only two possible values [38]. Independent variables may be categorical, discrete, or continuous, according to the modeling technique. Classification techniques include discriminant analysis [63], discriminant coordinates, the discriminative power technique [106], logistic regression [3], classification trees [102], pattern recognition [9],, artificial neural networks [68], case-based reasoning [5, 67], and fuzzy classification [21].

This section is based on [52, 54]. In software quality modeling, we usually consider a module as an "observation". In many of our studies [52], we use the groups *not fault-prone* and *fault-prone*; other groups may be used in other circumstances. A classification model predicts membership in one group or the other. However, since a model is not likely to be perfect, some modules will probably be misclassified by the model, compared to actual group membership. According to common statistics terminology, Type I errors misclassify modules that are actually not fault-prone as fault-prone. Type II errors misclassify modules that are actually fault-prone as not fault-prone.

We often model software development as a process that produces modules which are random samples from a large population of modules that might have been developed. From a Baysian viewpoint, our knowledge of the population is embodied in prior probabilities of class membership, i.e., "prior" to knowing the attributes of any modules in the sample. A classification technique, such as logistic regression, calculates the probability of being fault-prone based on module attributes, but this is not enough. A decision rule that minimizes misclassifications should also take into account the overall proportions of the underlying populations for each group, as well [42].

Let π_{fp} be the prior probability of membership in the fault-prone group, and let π_{nfp} be the prior probability of membership in the not fault-prone group. We want to choose prior probabilities that are appropriate for each set of modules that we classify. When a large fit data set is representative of the population, we choose the prior probabilities, π_{fp} and π_{nfp}, to be the proportion of fit modules in each group. Otherwise, we make adjustments according to our knowledge about the data set. Most software engineering classification models in the literature by other researchers use the uniform prior [3, 21, 68, 105, 108]. However, we have found that many techniques have poor accuracy on software metric data using uniform priors, because the proportion of fault-prone modules is in fact very small.

When judging quality of fit, we use priors based on the fit data set. When validating a model with an independent data set, we consider whether or not its proportion of fault-prone modules should be similar to the fit data set's. If so, we use prior probabilities based on the fit data set. When applying the model to other projects or subsequent releases (application data sets), we may adjust prior probabilities based on our knowledge of project attributes and plans.

In software engineering, the cost for acting on each type of erroneous prediction will depend on the process improvement technique that uses the prediction [46]. For example, suppose we apply a quality-enhancement processes, such as additional reviews, to modules identified as fault-prone. The cost of a Type I misclassification is to waste time on additional reviews of a module that is actually not fault-prone. The cost of a Type II misclassification is the lost opportunity to review a fault-prone module and detect its faults earlier in development. A fault that might have been discovered during a review will end up being discovered later in the project, when the cost of fixing it is

much greater.

Let C_I be the cost of a Type I misclassification, and let C_{II} be the cost of a Type II misclassification. A classification rule that minimizes the number of misclassifications may include prior probabilities, but this is not enough. Not all misclassifications are equivalent. Some types cost more than others. An optimal classification rule should minimize the expected cost of misclassifications (ECM), given by

$$ECM = C_I \pi_{\text{nfp}} \Pr(fp|nfp) + C_{II} \pi_{\text{fp}} \Pr(nfp|fp) \tag{2}$$

where $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$ are the Type I and Type II misclassification rates, respectively.

Given a classification model that estimates some measure of the likelihood of module i being fault-prone, $f_{\text{fp}}(\mathbf{x}_i)$, and not fault-prone, $f_{\text{nfp}}(\mathbf{x}_i)$, module i can be classified as fault-prone or not by a rule that minimizes the expected cost of misclassification.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{If } \frac{f_{\text{nfp}}}{f_{\text{fp}}} \geq \left(\frac{C_{II}}{C_I}\right)\left(\frac{\pi_{\text{fp}}}{\pi_{\text{nfp}}}\right) \\ \textit{fault-prone} & \text{Otherwise} \end{cases} \tag{3}$$

This rule minimizes the expected cost of misclassification (Equation (2)) as shown in Johnson and Wichern [42], and generalizes when priors or costs are equal. We have found that this rule is valuable in software engineering applications.

In many projects, costs of misclassification are difficult to estimate. Similarly, prior probabilities may also be unknown or difficult to estimate. The minimum-expected-cost rule in Equation (3) may be impractical in such cases. Thus, a more general rule that does not require these parameters is needed.

There is a tradeoff between $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$. We have observed that as one goes down, the other goes up. Our goal is to design a practical, flexible classification rule

that allows appropriate emphasis on each type of misclassification according to the needs of the project. The following rule enables a project to select the best balance between the misclassification rates.

$$Class(\mathbf{x}_i) = \begin{cases} \textit{not fault-prone} & \text{if } \frac{f_{\text{nfp}}(\mathbf{x}_i)}{f_{\text{fp}}(\mathbf{x}_i)} \geq \zeta \\ \textit{fault-prone} & \text{otherwise} \end{cases} \tag{4}$$

where ζ is a parameter which we can choose. We have observed in several empirical studies that the misclassification rates, which range in value from zero to one, are monotonic functions of ζ. We estimate these functions by repeated calculations with a *fit* data set. Given a candidate value of ζ, we estimate $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$. We repeat for various values of ζ. Determining the ζ that corresponds to a project's preferred balance between misclassification rates is straightforward when they are monotonic functions of ζ.

The generalized classification rule in Equation (4) does not depend on our knowledge of π_{nfp} and π_{fp}, nor of C_I and C_{II}, and thus, is useful when these quantities are difficult to estimate. However, if information is available, having selected a preferred ζ, one can interpret the value of ζ in terms of the priors ratio times the cost ratio, as in the minimum-expected-cost rule.

If one chooses ζ such that $\Pr(fp|nfp) = \Pr(nfp|fp)$, then the larger misclassification rate is minimized [107]. It is especially appropriate when $\pi_{\text{fp}} \ll \pi_{\text{nfp}}$, e.g., LeGall et al. [79] identified only 4% to 6% of modules as high-risk. In such cases, the maximum-correct-classification rule is often unsatisfactory. Equal misclassification rates may also be appropriate when C_{II}/C_I is so large that the minimum-expected-cost rule is unaffordable. In practice, we can achieve only approximate equality due to finite discrete data sets.

Order. We have learned from our industry collaborators that uncertain resource constraints may make classification models impractical. Consequently, we are developing empirical modeling techniques that yield rank-order by faults [48, 50]. Preliminary results have produced robust empirical models and innovative model evaluation criteria. Building on previous research that discussed ordering modules [29, 71, 94], we are pursuing other practical uses for such models. The following is based on [50].

Predicting the exact number of faults in each module is often not necessary; previous research has focused on classification models to identify *fault-prone* and *not fault-prone* modules [8, 21, 56, 63, 88]. Such models require that *fault-prone* be defined before modeling, usually via a threshold on the number of faults expected. However, due to uncertain resource constraints that limit the amount of reliability-improvement effort, software development managers often cannot choose an appropriate threshold at the time of modeling. In such cases, a prediction of the rank-order of modules, from the least to the most fault-prone, is more useful [71, 94]. With a predicted rank-order in hand, one can select as many from the top of the list for reliability enhancement as resources will allow.

A *module-order model* [55] predicts the rank-order of modules according to a quantitative quality factor, such as the number of faults. A module-order model has an underlying quantitative software quality model as the basis for predictions, without specifying *a priori* a threshold to define *fault-prone*. We evaluate module-order models from a project management perspective.

Our objective is to develop a model that will predict the relative quality of each module, especially those which are most fault-prone. In particular, we are interested in the order of modules according to the number of faults. Even though the number

of faults is absolute scale when directly measured, our recommended use of a predicted value is only ordinal scale [14]. Classification models, on the other hand, treat dependent variables as nominal scale, such as whether a module is *fault-prone* or not. A module-order model consists of the following components.

1. An underlying quantitative software quality model.

2. A ranking of modules according to a quality measure predicted by the underlying model.

3. A procedure for evaluating the accuracy of a model's ranking.

Suppose we have a quantitative model, $F_i = f(\mathbf{x}_i)$, where the number of faults, F_i, in module i is a function of its software measurements, the vector $\mathbf{x}_i$. Let $\widehat{F}(\mathbf{x}_i)$ be the estimate of F_i by a fitted model, $\hat{f}(\mathbf{x}_i)$. We use a simple quantitative modeling technique, multiple linear regression, without resorting to sophisticated estimation techniques [70]. Other quantitative modeling techniques could be used, such as nonlinear regression [66], regression trees [32], or computational intelligence techniques [9, 30, 72, 80]. Future research will investigate the effects of such refinements.

A simple underlying quantitative modeling technique is an advantage for module-order modeling. Users of module-order models with familiar underlying modeling techniques can easily see that the results were derived in a reasonable manner. Consequently, the module-order modeling technique lends itself to user acceptance. Module-order models are not "black boxes".

Let R_i be the percentile rank of observation i in a perfect ranking of modules according to F_i. Let $\widehat{R}(\mathbf{x}_i)$ be the percentile rank of observation i in the predicted ranking according to $\widehat{F}(\mathbf{x}_i)$.

Spearman correlation between actual and predicted rankings is a conventional approach to evaluating the accuracy of a module-order model [95, 109]. Spearman correlation evaluates the exact order over the entire set of modules. It is not appropriate for our application, because we do not care whether the rankings match exactly. Within the group that is enhanced, the order they are enhanced does not matter. They get equal treatment. Similarly, for those modules not enhanced, order does not matter. What does matter is where a module falls in relationship to a cutoff percentile that marks the end of the sequence of enhanced modules. However, at the time of modeling, we are uncertain what the cutoff will be. We want the reliability enhancement processes to find as many faults as possible. A module-order model enables management to enhance modules in the predicted order, confident that the total number of faults found will be close to expectations, even though the order of enhancement may not be perfect. Based on these considerations, the following is our evaluation procedure for a module-order model. Given a model, and a validation data set indexed by i:

1. Management will choose to enhance modules in priority order, beginning with the most fault-prone. However, the rank of the last module enhanced is uncertain at the time of modeling. Determine a range of percentiles that covers management's options for the last module, based on the schedule and resources allocated to reliability enhancement and associated uncertainties. Choose a set of representative cutoff percentiles, c, from that range.

2. For each cutoff percentile value of interest, c, define the number of faults accounted for by modules above the percentile c:

$$G(c) = \sum_{i:R_i \geq c} F_i \tag{5}$$

$$\widehat{G}(c) = \sum_{i:\widehat{R}(\mathbf{x}_i) \geq c} F_i \tag{6}$$

where higher c corresponds to the more fault-prone modules.

3. Let G_{tot} be the total number of actual faults in the validation data set's software modules. Calculate the percentage of faults accounted for by each ranking, namely, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$, and depict the accuracy of the model with an Alberg diagram [94].

4. Calculate a function measuring model performance, $\phi(c)$, which indicates how closely the faults accounted for by the model ranking match those of the perfect ranking.

$$\phi(c) = \frac{\widehat{G}(c)}{G(c)} \tag{7}$$

Plot performance as a function of c.

We want to identify only the worst modules, because resources are usually limited for reliability enhancement. In our example [50], we chose 50 through 95 percentiles, in 5 percent increments as a hypothetical preferred set of cutoff percentiles. If resources for reliability enhancement are indeed limited, it is unlikely that more than 50% of the modules will be enhanced. Another project might choose different percentiles, but this set illustrates our methodology. Cutoff percentiles can be calculated easily, if management's objective is in terms of the number of modules enhanced.

Ohlsson and Alberg introduce a variation of Pareto diagrams which they call "Alberg diagrams" [94]. Curves are plotted for an ordering based on the actual number of faults, and an ordering based on the number of faults predicted by a model. We employ Alberg diagrams [94] in this paper as an informal depiction of model accuracy.

In our context, an Alberg diagram consists of two curves, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$ plotted as functions of the percentage of modules selected for reliability enhancement, i.e., $1-c$. Modules are selected for reliability enhancement in descending order, beginning with the most fault-prone. The percentages of all faults, $G(c)/G_{tot}$ and $\widehat{G}(c)/G_{tot}$, give us insight into the importance of each cutoff percentile. If the two curves are close together, then the model is considered accurate.

When model predictions are intended only for ordinal purposes, Ohlsson et al. [94, 95], consider the model with the smallest area between curves to be the most useful. In case studies, they also compared models by the distance between the curves at selected percentages of modules.

The percentage of faults in a perfect ranking, $\phi(c)$, indicates the accuracy of the model at a given c. The variation in $\phi(c)$ over a range of c indicates the robustness of the model; small variation implies a robust model. A robust model is important in the face of uncertain resources for reliability enhancement. Because the number of cutoff percentiles is small, a statistical measure of robustness should be a simple measure of variation, such as the range of $\phi(c)$. We prefer a graphical presentation.

Quantity. A quantitative model is an equation (or algorithm) where the dependent variable is a quantity that is a function of one or more independent variables. If one supplies values for the independent variables, then one can calculate the value of the dependent variable. The following is based on [65].

Even though we may have a long list of candidate independent variables, it is possible that some do not significantly influence the dependent variable. If an insignificant variable is included in the model, it may add noise to the results and may cloud interpretation of

the model. For example, if a coefficient, a_j for the j^{th} variable in a linear model is not significantly different from zero, then it is best to omit that term from the model. The process of determining which variables are significant is called *model selection.*

Each kind of mathematical model has its own techniques for selecting significant variables. The following is an iterative statistical approach for multiple linear regression. Having specified a list of candidate independent variables, variables are entered into the model in an incremental manner, based on an F test from analysis of variance which is recomputed for each change in the current model. Begin with no variables in the model. Add the variable not already in the model with the best significance level, as long as its significance is better than the threshold. Then remove the variable already in the model with the worst significance level, as long as its significance is worse than the threshold. Repeat these steps until no variable can be added to the model

Many models have a general mathematical form with parameters that must be chosen so that the *fit* data set matches the model as closely as possible. This step consists of estimating the values of such parameters.

When using a mathematical model, the parameters are known, the independent variable values are known, and one then calculates dependent variable values. In contrast, when estimating model parameters, the independent variable values are known, the dependent variable values are known, and one must solve for the best estimate of the unknown parameter values by minimizing some overall measure of error.

For example, suppose we use multiple linear regression on the *fit* data set to estimate model parameters. A multivariate linear model is an equation where the dependent variable, y, is a linear function of the independent variables, $x_1, \ldots, x_p$. Suppose there are n observations in the fit data set, and the subscript i indicates data for the i^{th}

observation. In general, a multivariate linear model has the following form.

$$\hat{y}_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} \quad (8)$$

$$y_i = a_0 + a_1 x_{i1} + \ldots + a_p x_{ip} + e_i \quad (9)$$

where $x_{i1}, \ldots, x_{ip}$ are the independent variables' values, $a_0, \ldots, a_p$ are parameters to be estimated, $\hat{y}_i$ is the predicted value of the dependent variable, y_i is the dependent variable's actual value, and $e_i = y_i - \hat{y}_i$ is the error for the i^{th} observation. In an example software quality model, y is the number of faults and the independent variables are software measurements. We estimate the parameters, $a_0, \ldots, a_p$, using the *least squares* method. This method chooses a set of parameter values that minimizes $\sum_{i=1}^{n} e_i^2$ [107].

9.4 Model evaluation

Outliers. In statistical modeling, a parameter estimation technique may be overly sensitive to a few unusual observations in the fit data set, so that the model does not accurately characterize the overall population of observations. Consequently, when applied to an independent data set drawn from the population, the model may not predict accurately overall. The analyst may decide that the few overly influential observations should be excluded from the fit data set, because they are not members of the population that the model is intended to represent. Sometimes such observations turn out to be due to data-collection problems, rather than true anomalies.

Each modeling technique has its own methods for detecting outliers. For example, consider multiple linear regression [65]. Unfortunately, the least squares method is unduly influenced by unusual data points, *outliers*. Such points may be inappropriate for the model, and thus, we do not use outliers when estimating the final parameters. We

detect outliers using the *R-Student* statistic [92]. An iterative process detects outliers, removes them from the fit data set, fits the model again with the reduced fit data set, and calculates *R-Student* statistics again, until there are no more outliers at the given significance level. After all outliers have been removed from the fit data set, the final parameters are estimated.

Unbiased estimates of accuracy. The parameters of a statistical model are estimated using a fit data set. It is tacitly assumed that the observations in the fit data sets are a fair representation of the population. Similarly, when a statistical model is evaluated, it is assumed that the observations in the test data set are statistically independent of the fit data set. When these assumptions are violated, the estimate of model accuracy may be *biased*. In other words, the evaluation may be more optimistic than it should be. An unbiased estimate of model accuracy is important in our context, because one intends to use model predictions to guide software development activities. A biased evaluation would give a false sense of security to the software developers. The following is based on [47].

Each of the methods described below [20, p.392] evaluates the accuracy of a model by predicting the dependent variable of each observation in the test data set, and then calculating the accuracy. However, each method uses a different test data set.

Resubstitution. This method uses the fit data set as a test data set. Because the fit and test data sets are not at all independent, this is the least realistic of the methods discussed here. After model parameters are estimated, the dependent variable of each observation in the fit data set is predicted. The accuracy characterizes "model fit". This assessment of model accuracy is often overly optimistic.

Subsequent project. This method uses one project as a fit data set and a subsequent similar project as a test data set. This is a realistic simulation of applying a software quality model in practice. This method is sensitive to the degree of similarity between the projects. One must address the question, "Is it appropriate to model the observations in both projects as coming from the same population?" This same question must be answered when applying the model in practice, as well.

Data splitting. This is sometimes called the "Holdout" method [31]. Fit and test data sets are derived from a single data set by impartially sampling from available observations. This is also a simulation of applying a model in practice. Data splitting may be appropriate when data on a similar subsequent project is not available. The proportion of observations in each data set is chosen according to sample sizes needed for the various statistical techniques to be employed. Consequently, this method often requires large samples of available data. Statistical similarity of the fit and test data set is assured by the partitioning method. If statistical variation in the partitioning is a concern, then a number of data set pairs can be generated by random resampling, and then analyzed [47].

Cross-validation. This is sometimes called the "U-Method" [22, 78]. Suppose there are n observations available. Let one observation be the test data set and all the others be the fit data set. Build a model, and evaluate it for the current observation. Repeat for each observation, resulting in n models. Let the accuracy summarize the n evaluations of the models. In contrast to resubstitution, this does not have serious bias. This method is appropriate for smaller data sets than data splitting, but involves more computation per observation.

A variation of the cross-validation approach partitions the available data into disjoint

test data sets. For example, if each test data set has one tenth of the observations, then we generate $\nu = 10$ models, using the remaining nine tenths as a fit data set. This is called "ν-fold cross-validation" [32]. Since the test data sets do not have observations in common, they are statistically independent.

Overfitting. One's expectation for model accuracy is generally formed by experience with the training data set. *Overfitting* is characterized by an adverse deviation from that base of experience. An overfitted model reflects the structure of the training data set too closely. Even though a model appears to be accurate on training data, if overfitted, the accuracy estimated by resubstitution may be biased. Moreover, a software engineering interpretation of an overfitted software quality model's structure may not be true for similar systems or subsequent releases.

The focus of this section is classification models of software quality based on software metrics. For example, neural network models and classification-tree models are often vulnerable to overfitting. The following is based on [51].

In the statistics literature, overfitting is often defined as *bias*, namely, the difference between a model's actual misclassification rate and its purported rate [110]. Overfitting can be detected by a statistically unbiased estimate of model accuracy compared to its possibly biased accuracy on training data. The difference is a measure of the degree of *overfitting*. The total number of misclassifications can be a satisfactory measure of a model's accuracy if the proportions of each class are approximately equal and the two kinds of costs of misclassification are about equal. However, in software engineering applications, the proportion of fault-prone modules is often small and the practical penalty for misclassifying a fault-prone module as not fault-prone can be quite serious after re-

lease. Thus, we prefer the expected cost of misclassification as a measure of a model's accuracy.

In software engineering practice, the penalty for a Type II misclassification is often much more severe than for a Type I. A software enhancement technique, such as extra reviews, typically has modest direct cost per module. The cost of a Type I misclassification, C_I is the effort wasted on a not fault-prone module. On the other hand, the cost of a Type II misclassification, C_{II}, is the lost opportunity to correct faults early. The consequences of letting a fault go undetected until after release can be very expensive indeed. We model the costs of misclassifying a module, C_I and C_{II}, as constants [47]. Future research will consider more sophisticated cost functions. The expected cost of misclassification of one module, ECM, takes prior probabilities and costs of misclassification into account.

$$ECM = C_I \Pr(fp|nfp)\,\pi_{\text{nfp}} + C_{II} \Pr(nfp|fp)\,\pi_{\text{fp}} \tag{10}$$

As a measure of overfitting, we propose the difference between expected costs of misclassification estimated by an independent evaluation data set and a training data set, normalized by the cost of a Type I misclassification. By Equation (10),

$$\Delta ECM = \frac{1}{C_I}(ECM_{\text{eval}} - ECM_{\text{train}}) \tag{11}$$

$$\begin{aligned} \Delta ECM &= \pi_{\text{nfp}}(\Pr(fp|nfp)_{\text{eval}} - \Pr(fp|nfp)_{\text{train}}) \\ &+ \zeta\,\pi_{\text{nfp}}(\Pr(nfp|fp)_{\text{eval}} - \Pr(nfp|fp)_{\text{train}}) \end{aligned} \tag{12}$$

where subscripts indicate the training data set (train) and an evaluation data set (eval), and ζ is per the classification rule in Equation (4) above. In our application, the unit of measure is the cost of misclassifying a not fault-prone module. If ΔECM is positive,

then the accuracy of the model on the training data set is somewhat misleading. When ΔECM is zero, we have avoided overfitting. If ΔECM is negative, then the accuracy of the model on the evaluation data set is better than on the training data set, a pleasant surprise.

9.5 Model utilization

When the parameters have been estimated, and given each set of independent variable values, a model can calculate a value of the dependent variable. Since the independent variables are known earlier than the actual value of the dependent variable, the calculated value is a prediction.

10 What tools are available?

10.1 Configuration management systems

A *configuration management system* is an information system for managing multiple versions of artifacts produced by software development processes. For example, most configuration management systems support storage and retrieval of versions of source code. Other features may regulate changes to source code, so that team members do not interfere with each other, and record the history of changes for later review. Many configuration management systems support storage of other development artifacts also. A configuration management system is a major source of artifacts to measure (product metrics) and of process metrics.

10.2 Problem reporting systems

A *problem reporting system* is an information system for managing software faults from initial discovery through distribution of fixes. In other words, it records events in the debugging process. Most developers of large software products use such systems. A problem reporting system is a major source of quality metrics and process metrics.

10.3 SATC's tools

The Software Assurance Technology Center (SATC) at NASA Goddard Space Flight Center has been collecting code metrics for over seven years. In their data base, metrics are aggregated to the project level.[3]

Non-object-oriented metrics are available for the following languages [104].

- C
- C++ (non-object-oriented viewpoint)
- FORTRAN
- Ada
- Omnis
- Jovial

Object-oriented metrics are available for the following languages [104].

- C++
- Java

[3]For more information, see http://satc.gsfc.nasa.gov/

Non-OO metrics.

#Modules Total number of modules in a project

LOC Total lines of code (source+blank+comments)

Blank Total number of blank lines

%Comment $Comm/(LOC - Blank)$

ExecStmts Total number of executable statements

Comm Total number of comments (freestanding+inline)

#goto Total number of GOTOs

#sql Total number of SQL calls

CyclomaticComplexity Number of linearly independent test paths (McCabe)

ExtendedCyclomaticComplexity Similar to cyclomatic complexity extended by taking into account compound decisions

OO metrics.

#File Total number of files

#Classes Total number of classes

#NTLC Total number of top level classes

LOC Total lines of code (source+blank+comments)

Preprocess Total number of preprocessor lines (Imports) (e.g. #include)

Blank Total number of blank lines

Comm Total number of comments (freestanding+inline)

CP Comment Percentage, $Comm/(LOC - Blank)$

NCNB Non-comment non-blank, also known as source lines of code

ExecStmts Total number of executable statements

NOM Number of methods

WMC Weighted methods per class. Sum of cyclomatic complexities of the methods in a class

CBO Coupling between objects. Number of other classes whose methods or instance variables are used by methods of this class

RFC Response for a class. Number of methods in the class plus number of methods called by each of these methods, where each called method is counted once.

10.4 Datrix

Datrix is a software analyzer product developed by Bell Canada in collaboration with others [82]. It is avaliable for quick trials, university researchers, and commercial users under licence.[4]

These metrics are available for the following languages [18].

- C
- C++
- Java

Routine metrics.

RtnArgXplSum Sum of the explicit argument numbers (actual parameters) passed to other function by all the explicit function calls made in the routine

RtnCalXplNbr Number of explicit function/method calls in the routine

RtnCastXplNbr Number of explicit type casts in the routine

RtnComNbr Number of comment sections in the routine scope (between the routine brackets {...})

RtnComVol Size in characters of all the comments in the routine, without considering the comments within nested classes or routines

[4]Information on Datrix is available at http://www.iro.umontreal.ca/labs/gelo/datrix/

RtnCplCtlAvg The mean control predicate complexity

RtnCplCtlMax The maximal control predicate complexity

RtnCplCtlSum Total (sum) complexity of the control predicates (test expressions) composing the decision and loop statements within the routine

RtnCplCycNbr Cyclomatic number of the routine

RtnCplExeAvg The mean executable statement complexity

RtnCplExeMax The maximal executable statement complexity

RtnCplExeSum Total (sum) complexity of the executable statements within the routine

RtnStmDecRtnNbr Number of function/routine declaration statements within the routine

RtnStmDecObjNbr Number of variable/object declaration statements in the routine

RtnStmDecPrmNbr Number of parameters of the routine

RtnStmDecTypeNbr Number of type/class declaration statements in the routine

RtnLnsNbr Number of lines of the routine

RtnLnsSkpSum Number of full or partial lines skipped in the routine, due to syntax errors

RtnScpNbr Number of scopes within the scopes of the routine

RtnScpNstLvlAvg Average nesting level of the scopes in the routine

RtnScpNstLvlMax Maximal nesting level in the routine

RtnScpNstLvlSum Sum of nesting level values for all scopes in the routine

RtnStmCtlBrkNbr Number of break statements in the routine

RtnStmCtlCaseNbr Number of C-language case-like statements in the routine

RtnStmCtlNbr Number of control-flow statements in the routine

RtnStmCtlThwNbr Number of throw (cast) statements in the routine

RtnStmCtlCtnNbr Number of continue statements in the routine

RtnStmDecNbr Number of declarative statements in the routine

RtnStmCtlDfltNbr Number of default statements in the routine

RtnStmExeNbr Number of executable statements in the routine

RtnStmCtlGotoNbr Number of goto statements in the routine

RtnStmCtlIfNbr Number of if statements in the routine

RtnLblNbr Number of label statements in the routine

RtnStmCtlLopNbr Number of loop statements in the routine

RtnStmNbr Number of statements in the routine

RtnStmNstLvlAvg Average nesting level of statements in the routine

RtnStmNstLvlSum Sum of nesting level values of each statement in of the routine

RtnStmCtlRetNbr Number of return statements in the routine

RtnStmCtlSwiNbr Number of C-language switch-like constructs in the routine

RtnStmXpdNbr Expanded statement number: size (in number of statements) of the routine after expansion (limited loop unfolding operation)

RtnStxErrNbr Number of syntax error that occurred while parsing the routine

Class metrics.

ClaAttFinNbr Number of final attributes in the class

ClaAttHidRto Ratio of attribute hiding for the class

ClaAttNbr Number of (instance/variable) attributes in the class

ClaAttPriNbr Number of private attributes in the class

ClaAttProNbr Number of protected attributes in the class

ClaAttPubNbr Number of public attributes in the class

ClaAttPtrNbr Number of (instance/variable) attributes of pointer type in the class

ClaAttStaNbr Number of static attributes in the class

ClaClaNstNbr Number of nested classes, where a nested class is a class defined within the scope of another class

ClaComNbr Number of comment sections in the class scope (between the class brackets {...}), without considering the comments within nested classes or routines

ClaComVol Size in characters of all the comments in the class, without considering the comments within nested classes or routines

ClaImpIntNbr Number of interfaces implemented by the class

ClaInhDirNbr Number of direct parent classes found in the first level of inheritance of that class

ClaInhIndNbr Total number of parent classes found in all inheritance levels of that class

ClaInhLvlMax Inhertance level of that class

ClaLnsNbr Number of lines of that class

ClaLnsSkpSum Number of full or partial lines skipped in the class, including empty lines, due to syntax errors

ClaMetFinNbr Number of final methods in the class

ClaMetNbr Number of methods in the class

ClaMetPriNbr Number of private methods in the class

ClaMetProNbr Number of protected methods in the class

ClaMetPubNbr Number of public methods in the class

ClaMetPurNbr Number of pure virtual methods in the class

ClaMetStaNbr Number of static methods in the class

ClaMsgNbr Number of messages in the class, where a "message" is considered to be a way to access the class, e.g. a class attribute or a class method

ClaNamLen Length of the class name

ClaStxErrNbr Number of syntax error that occurred while parsing the class

File metrics.

FilComGlbNbr Number of comment sections in the global scope of a file (thus excluding the comments inside routine or class scopes within the file)

FilComGlbVol Size in characters of all the comments in the global scope of a file (without considering the comments inside routine or class scopes within the file)

FilComTotNbr Total number of comment sections in the file (considering the comments inside routine or class scopes within the file)

FilComTotVol Size in characters of all the comments in the file (considering the comments inside routine or class scopes within the file)

FilDecClaNbr Number of classes declared within the file

FilDecGncTypNbr Number of generic classes (template class declaration) declared within the file

FilDecGndTypTotNbr Total number of generated classes (template class instances) declared within the file

FilDecStruNbr Number of struct types declared within the file

FilDecObjExtNbr Number of extern objects declared within the global scope of the file

FilDefObjGlbNbr Number of global variables/objects defines within the global scope of the file

FilDefRtnNbr Number of functions/routines defined within the file

FilIncNbr Total number of files included by the current file

FilIncDirNbr Total number of files directly included by the current file

FilLnsNbr Number of lines of the file

FilLnsSkpSum Number of full or partial lines skipped in the file, due to syntax errors

FilStxErrNbr Number of full or partial lines skipped in the file, due to syntax errors

Halstead metrics.

OpdNbr Total number of operands, N_2

OpdUnqNbr Number of distinct operands, n_2

OprNbr Total number of operators, N_1

OperUnqNbr Number of distinct operators, n_1

HalDif Difficulty, D

HalEff Effort, E

HalLen Length, N

HalLvl Level, L

HalVoc Vocabulary, n

HalVol Volume, l

10.5 EMERALD

Enhanced Measurement for Early Risk Assessment of Latent Defects (EMERALD) is a sophisticated system of decision support tools used by software designers and managers to assess risk and improve software quality and reliability. It was developed by Nortel Networks in partnership with Bell Canada and others [39]. The EMERALD team performs services for projects within Nortel Networks and for outside clients.[5]

The following are available data elements on a per module basis [49].[6]

[5] Further information about EMERALD is available from John P. Hudepohl, Nortel Networks, Research Triangle Park, North Carolina. Email: hudepohl@nortelnetworks.com

[6] *aka* means "also known as".

Module identifiers.

STREAM Designation of system release.

MODULE Name of software unit. Basic unit for reported measurements.

DRU Development release unit. Name for architecture layer.

RELEASE The *DRU* release number for this module.

LOAD The load identifier for this module.

ARCHID Architecture group name.

Data provided by Datrix. EMERALD uses the Datrix metric analyzer, among others, to collect metrics from source code.

Process data. The following data items were derived from the Nortel Networks problem reporting system and configuration management system.

PRS_PRS The number of internal problems found in this module during development against the current release.

VO_PRS The number of beta-test problems found in this module during beta testing of the latest release. (aka *BETA_PR*)

CUST_PRS The number of field (external) problems found in the module against the current release. (aka *Faults*)

UPDIND Indicator as to whether or not the module changed during development cycle. ($UPD > 0$ means changed)

PRS The total number of problems fixed during the development cycle (stream). (aka *TOT_FIX*)
$PRS = INT + VO + EXT$.

INT Total number of different problems that were fixed during current stream where the problems originated from issues found by designers. (aka *DES_FIX*)

VO Total number of different problems that were fixed during current stream where the problems originated from issues found by beta testing. VO stands for Verification Office. (aka *BETA_FIX*)

EXT Total number of different problems that were fixed during current stream where the problems originated from issues found by external customers. (aka *CUST_FIX*)

FTUPD Total number of changes to the code for new feature reasons. (aka *REQ_UPD*)

UPD Total number of changes to the code for any reason. (aka *TOT_UPD*)

DIFFFT Number of different feature identifiers used to change the module during current stream. (aka *REQ*)

SGRTHTOT Net increase in source lines of code due to software changes. (aka *SRC_GRO*)

SMODTOT Net new and changed source lines of code due to software changes. (aka *SRC_MOD*)

UNQUPDID Number of different designers creating changes for this module. (aka *UNQ_DES*)

VLOUPDSM Number of updates for any reason to this module by designers who had 10 or less total updates in entire Nortel career. (aka *VLO_UPD*)

LOUPDSM Number of updates for any reason to this module by designers who had 20 or less total updates in entire Nortel career. (aka *LO_UPD*)

AVGUPDNO The average number of updates that a designer has had in their Nortel career when they updated this module for any reason.

Execution data.

USAGE A rough measure of the deployment percentage of the module. Higher values imply more widely deployed and used.

OPSCORE Operational Profile score on usage/criticality based on Bellcore study.

RESCPU Execution time (microseconds) of an average call on a switch serving residential customers.

BUSCPU Execution time (microseconds) of an average call on a switch serving business customers.

TANCPU Execution time (microseconds) of an average call on a tandem switch.

Other data.

AIM Abductive Inference Machine — A composite complexity score.

TTRISK A composite complexity index made up of AIM and other factors.

Complexity level. LEVEL1, *LEVEL2*, *LEVEL3*, *LEVEL4*, and *LEVEL5* each count the number of procedures at each level of the Datrix complexity model.

OUTRANGE Number of routines with metrics out of acceptable range, according to DATRIX thresholds.

Halstead metrics. HALDIF, HALEFF, HALLEN, HALLVL, HALVOC, and *HALVOL* are synthetic metrics defined by Halstead [33].

SSIZE Size of module (lines of code). (This was inconsistent with Datrix *LOC*.)

Synthetic variables. As part of FAU's collaboration with the EMERALD team, we defined the following variables.

TOT_PRS The total number of problems found in this module during development against the current release. $TOT\_PRS = PRS\_PRS + VO\_PRS + CUST\_PRS$

FILINC2 The number of second and following file includes.
$FILINC2 = FILINC - FILINCUQ$

CAL2 The number of second and following calls to others. $CAL2 = CAL - CALUNQ$

CNDCPLSM The total conditional arc complexity.
$CNDCPLSM = CNDCPLAV \times CND$

CNDSPNSM The total span of conditional arcs.
$CNDSPNSM = CNDSPNAV \times CND$

CNDNOT The number of non-conditional arcs. $CNDNOT = ARC - CND$

IFTH The number of non-loop conditional arcs. $IFTH = CND - LOP$

LOPVOL The amount of control structure volume in loops.
$LOPVOL = LOPSTRAV \times CTRVOL$

LGPATH The base two logarithm of the number of independent paths. In a case study, the maximum value of *PTHIND* was 3.8E37, which was too large for statistical processing. $LGPATH = \log_2 PTHIND$

NDSINT The number of internal nodes.
$NDSINT = NDS - NDSENT - NDSEXT - NDSPND$

STMCPLSM The total amount of statement complexity.
$STMCPLSM = STMCPLAV \times STM$

VARLENSM The total length of variables.
$VARLENSM = VARLENAV \times VARUSDUQ$

VARUSD2 The number of second and following uses of variables.
$VARUSD2 = VARUSD - VARUSDUQ$

LO2UPDSM Number of updates for any reason to this module by designers who had between 11 and 20 total updates in entire Nortel career.
$LO2UPDSM = LOUPDSM - VLOUPDSM.$

UPDNOSM The total number of updates that designers had in their Nortel careers when they updated this module for any reason.
$UPDNOSM = AVGUPDNO \times UPD$ (aka UPD_CAR)

10.6 Logiscope

Logiscope RuleChecker is a commercially available software analyzer product of Telelogic AB.[7] Metrics are available for the following languages, and are listed separately.

- C [112]
- C++ [113]

C metrics.

Textual complexity metrics.

STMT Number of statements

n1 Number of distinct operators

N1 Number of operator occurrences

[7] Information on Logiscope RuleChecker is available at http://www.telelogic.com/logiscope/

n2 Number of distinct operands

N2 Number of operand occurrences

PR_LGTH Program length, N

AVGS Average size of statements, $AVGS = (N1 + N2)/STMT$

VOCF Vocabulary frequency, $(N1 + N2)/(n1 + n2)$

Structural complexity metrics.

RETU Number of return statements

N_IN Number of entry nodes

N_OUT Number of exit nodes

NB_IO Number of entry and exit nodes, $NB\_IO = N\_IN + N\_OUT$

GOTO Number of GOTO statements

COND_STRUCT Number of specific branchings (e.g. BREAK or CONTINUE)

NEST Maximum number of nesting levels

VG Cyclomatic number, $VG = E - N + 1$

PATH Number of paths

Data complexity metrics.

UPRO Number of unknown prototypes

MACP Number of time macro instructions with parameters are used

MACC Number of time macro instructions (constants) are used

MAC Number of macro instructions, $MAC = MACP + MACC$

CALL Number of function calls

CALL_PATHS Number of different function calls

LVAR Number of local variables

PARA Number of parameters

Comment metrics.

N_COM Number of blocks of comments

BCOB Number of blocks of comments before the function

COM_R Number of blocks of comments per statement (comments rate)
$COM\_R = N\_COM/STMT$

LCOM Number of lines of comments

LCOB Number of lines of comments before the function

CCOM Number of characters in comments

CCOB Number of characters in comments before the function

C++ metrics.

Function-level metrics.

AVGA Sum of average size of instructions

BCOB Number of blocks of comments before the function

BCOM Number of blocks of comments in the function

CALL_PATHSe Number of Distinct calls to functions defined outside the class

CALL_PATHSi Number of Distinct calls to functions defined in the class

CALLe Number of calls to functions defined outside the class

CALLi Number of calls to functions defined in the class

CCOB Number of characters in comments before the function

CCOM Number of characters in comments

COND_STRUCT Number of specific branchings (e.g. BREAK or CONTINUE)

GOTO Number of GOTO statements

LCOB Number of lines of comments before the function

LCOM Number of lines of comments

LEVL Number of levels

LMABS Number of abstract methods

LVAR Number of local variables

LVARop Number of class type local variables

MACC Number of time macro instructions (constants) are used

MACP Number of time macro instructions with parameters are used

N1 Number of operator occurrences

n1 Number of distinct operators

N2 Number of operand occurrences

n2 Number of distinct operands

N_CDD Number of decision-to-decision paths

N_EXCEPT Number of exception handlers

N_GEN Number of generic parameters

N_RAISE Number of exception raise's

N_STRUCT Number of decisions

NBCALLING Number of calling's

NCONST Number of CONST declared variables

NINLINE Number of INLINE functions

PARAadd Number of parameters passed by address

PARAc Number of class type parameters

PARAval Number of parameters passed by value

PATH Number of paths

RETU Number of return statements

STMT Number of statements

U_PARA Number of used parameters

VAR_PATHSe Number of distinct uses of external attributes

VAR_PATHSi Number of distinct uses of local attributes

VARe Number of times external attributes are used

VARi Number of times local attributes are used

VG Cyclomatic number, $VG = E - N + 1$

Class-level metrics.

BCOBc Number of comment blocks before the class

BCOMc Number of comment blocks in the class

CCOBc Number of characters in comments before the class

CCOMc Number of characters in the class comments

COBC Coupling between classes

LACT Sum of class-type attributes of the class

LAPI Number of private attributes of the class

LAPO Number of protected attributes of the class

LAPU Number of public attributes of the class

LCOBc Number of lines of comments in the class

LCOMc Number of lines of comments before the class

LMCALL_PATHSe Total number of *CALL_PATHSe* from class methods

LMCALL_PATHSi Total number of *CALL_PATHSi* from class methods

LMCALLING Sum of methods' *NBCALLING*

LMDE Sum of methods defined in the class

LMPARA Sum of method parameters

LMPI Sum of private methods in the class

LMPIPATH Sum of *PATH* for private methods in the class

LMPO Sum of protected methods in the class

LMPIPATH Sum of *PATH* for protected methods in the class

LMPU Number of public methods in teh class

LMPUPATH Sum of *PATH* for public methods in the class

LMPUPARA Sum of public method parameters

LMRE Sum of methods redefined in the class

LMU_PARA Sum of method *U_PARA*

LMVG Sum of method *VG*

LMVAR_PATHSe Total number of *VAR_PATHSe* used by class methods

LMVAR_PATHSi Total number of *VAR_PATHSi* used by class methods

MII Number of class parents

N_GENc Number of generic parameters in the class

NMD Number of dependent methods

NOC Number of class children

System-level metrics.

AVGA Sum of average size of instructions

CBO Coupling between objects

GA_CYCLE Call graph recursions

GA_EDGE Number of edges in the call graph

GA_LEVL number of levels in the call graph

GA_MAX_DEG Maximum number of calling/called functions

GA_MAX_IN Maximum number of callings

GA_MAX_OUT Maximum number of called functions

GA_NODE Number of call graph nodes

GA_NSP Number of call graph roots

GA_NSS Number of call graph leaves

GH_CPX Hierarchical complexity of inheritance graph

GH_EDGE Number of edges in teh inheritance graph

GH_LEVL Number of levels in the inheritance graph

GH_MAX_DEG Maximum numer of inherited/derived classes

GH_MAX_IN Maximum number of derived classes

GH_MAX_OUT MAximum number of inherited classes

GH_NODE Number of classes in the inheritance graph

GH_NSP Number of leaf classes

GH_NSS Number of basic classes

GH_PC Protocol complexity of the inheritance graph

GH_URI Number of repeated inheritances

LCA Number of classes

LMA Number of functions

NMABS Number of abstract methods

NMM Sum of member fucntions

VGA Sum of VG's

Halstead derived metrics.

n Vocabulary, $n = n1 + n2$

N Length, $N = N1 + N2$

CN Estimated length, $CN = n1 \log_2 N1 + n2 \log_2 N2$

V Volume, $V = N \log_2 n$

V^* Potential volume

L Level, $L = V*/V$

I Intelligence content, $I = L\,V$

D Difficulty, $D = 1/L$

$LAMB$ Language level, $\lambda = L^2\,V$

E Effort, $E = V/L$

T Implementation time, $T = E/18$

B Potential errors, $B = E^{2/3}/3000$

10.7 FAU metrics analyzer for C

The Empirical Software Engineering Laboratory in the Department of Computer Science and Engineering, Florida Atlantic University, developed a research tool for collecting software metrics.[8] These metrics are available for ANSI C [44].

STMTS Number of executable statements

SEMIS Number of semicolons

ETA1 Number of unique oerators, η_1

N1 Total number of operators, N_1

ETA2 Number of unique operands, η_2

N2 Total number of operands, N_2

LOOPS Number of loops in the control flow graph

NODES Number of nodes in the control flow graph

[8] Further information is available from Dr. Taghi Khoshgoftaar, Department of Computer Science and Engineering, Florida Atlantic University, Boca Raton, Florida 33431. Email: taghi@cse.fau.edu

EDGES Number of edges in teh control flow graph

VG Cyclomatic complexity

VG2 Extended cyclomatic complexity, i.e. *VG* plus the number of logical operators

CALLSIN Number of calls to a function

CALLSOUT Number of calls out from the function

BAND Belady's bandwidth metric, which measure the average nesting level of the control flow graph

CALLSRCV Number of recursive calls

INPUTS Number of input parameters

OUTPUTS Number of output parameters (all pointer parameters are considered output)

DS Data structure complexity

GBLVARS Number of global references

Acknowledgments

We thank Ken McGill for his encouragement and support. We thank Bojan Cukic for helpful discussions. This work was supported in part by Cooperative Agreement NCC 2-1141 from NASA Ames Research Center, Software Technology Division (Independent Verification and Validation Facility). The findings and opinions in this paper belong solely to the authors, and are not necessarily those of the sponsor.

References

[1] W. W. Agresti and W. M. Evanco. Projecting software defects from analyzing Ada designs. *IEEE Transactions on Software Engineering*, 18(11):988–997, Nov. 1992.

[2] E. B. Allen and T. M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 119–127, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society.

[3] V. R. Basili, L. C. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct. 1996.

[4] V. R. Basili, R. W. Selby, and D. H. Hutchens. Experimentation in software engineering. *IEEE Transactions on Software Engineering*, SE–12(7):733–743, July 1986.

[5] Y. Berkovich. Software quality prediction using case-based reasoning. Master's thesis, Florida Atlantic University, Boca Raton, Florida USA, Aug. 2000. Advised by Taghi M. Khoshgoftaar.

[6] J. M. Bieman and B.-K. Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24(2):111–124, Feb. 1998.

[7] J. M. Bieman and L. M. Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20(8):644–657, Aug. 1994.

[8] L. C. Briand, V. R. Basili, and C. J. Hetmanski. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering*, 19(11):1028–1044, Nov. 1993.

[9] L. C. Briand, V. R. Basili, and W. M. Thomas. A pattern recognition approach for software engineering data analysis. *IEEE Transactions on Software Engineering*, 18(11):931–942, Nov. 1992.

[10] L. C. Briand, J. Daly, V. Porter, and J. Wüst. A comprehensive empirical validation of design measures for object-oriented systems. In *Proceedings Fifth International Software Metrics Symposium*, pages 246–257, Bethesda, MD USA, Nov. 1998. IEEE Computer Society.

[11] L. C. Briand, J. Daly, V. Porter, and J. Wüst. Predicting fault-prone classes with design measures in object-oriented systems. In *Proceedings the Ninth International Symposium on Software Reliability Engineering*, pages 334–343, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[12] L. C. Briand, J. W. Daly, and J. Wüst. A unified framework for cohesion measurement in object-oriented systems. In *Proceedings of the Fourth International Symposium on Software Metrics*, pages 43–53, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[13] L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan. 1999.

[14] L. C. Briand, K. El Emam, and S. Morasca. On the application of measurement theory in software engineering. *Empirical Software Engineering: An International Journal*, 1(1):61–88, 1996. See [15, 119].

[15] L. C. Briand, K. El Emam, and S. Morasca. Reply to "Comments to the paper: Briand, El Emam, Morasca: On the application of measurement theory in software engineering". *Empirical Software Engineering: An International Journal*, 2(3):317–322, 1997. See [14, 119].

[16] L. C. Briand, S. Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–85, Jan. 1996. See comments in [17, 101, 120].

[17] L. C. Briand, S. Morasca, and V. R. Basili. Response to: Comments on "Property-based software engineering measurement": Refining the additivity properties. *IEEE Transactions on Software Engineering*, 23(3):196–197, Mar. 1997. See [16, 101].

[18] B. Canada. *Datrix Metric Reference Manual.* Montreal, Quebec, Canada, version 4.0 edition, May 2000. For Datrix version 3.6.9.

[19] S. R. Chidamber, D. P. Darcy, and C. F. Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering*, 24(8):629–639, Aug. 1998.

[20] W. R. Dillon and M. Goldstein. *Multivariate Analysis: Methods and Applications.* John Wiley & Sons, New York, 1984.

[21] C. Ebert. Classification techniques for metric-based software development. *Software Quality Journal*, 5(4):255–272, Dec. 1996.

[22] B. Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, June 1983.

[23] W. M. Evanco and W. W. Agresti. A composite complexity approach for software defect modeling. *Software Quality Journal*, 3(1):27–44, Mar. 1994.

[24] U. M. Fayyad. Data mining and knowledge discovery: Making sense out of data. *IEEE Expert*, 11(4):20–25, Oct. 1996.

[25] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, Nov. 1996.

[26] N. E. Fenton. *Software Metrics: A Rigorous Approach.* Chapman and Hall, London, 1991.

[27] N. E. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, Mar. 1994.

[28] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, Sept. 1999.

[29] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, London, 2d edition, 1997.

[30] K. Ganesan, T. M. Khoshgoftaar, and E. B. Allen. Case-based software quality prediction. *International Journal of Software Engineering and Knowledge Engineering*, 9(6), 1999. In press.

[31] S. Geisser. The predictive sample reuse method with applications. *Journal of the American Statistical Association*, 70(350):320–328, June 1975.

[32] S. S. Gokhale and M. R. Lyu. Regression tree modeling for the prediction of software quality. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[33] M. H. Halstead. *Elements of Software Science*. Elsevier, New York, 1977.

[34] D. J. Hand. Data mining: Statistics and more? *The American Statistician*, 52(2):112–118, May 1998.

[35] R. Harrison, S. J. Counsell, and R. V. Nithi. An evaluation of the MOOD set of object-oriented software metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496, June 1998.

[36] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice Hall, Upper Saddle River, New Jersy USA, 1996.

[37] S. Henry and D. Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, SE–7(5):510–518, Sept. 1981.

[38] D. W. Hosmer, Jr. and S. Lemeshow. *Applied Logistic Regression*. John Wiley & Sons, New York, 1989.

[39] J. P. Hudepohl, S. J. Aud, T. M. Khoshgoftaar, E. B. Allen, and J. Mayrand. EMERALD: Software metrics and models on the desktop. *IEEE Software*, 13(5):56–60, Sept. 1996.

[40] IEEE Computer Society. *Proceedings: Sixth International Software Metrics Symposium*, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society Press.

[41] IEEE Computer Society. *Proceedings: Tenth International Symposium on Software Reliability Engineering*, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society Press.

[42] R. A. Johnson and D. W. Wichern. *Applied Multivariate Statistical Analysis.* Prentice Hall, Englewood Cliffs, NJ, 3d edition, 1992.

[43] W. D. Jones, J. P. Hudepohl, T. M. Khoshgoftaar, and E. B. Allen. Application of a usage profile in software quality models. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pages 148–157, Amsterdam, Netherlands, Mar. 1999. IEEE Computer Society.

[44] S. Jordan. Software metrics collection: Two new research tools. Master's thesis, Florida Atlantic University, Boca Raton, FL, Dec. 1997. Advised by Taghi M. Khoshgoftaar.

[45] T. M. Khoshgoftaar and E. B. Allen. Multivariate assessment of complex software systems: A comparative study. In *Proceedings of the First International Conference on Engineering of Complex Computer Systems*, pages 389–396, Fort Lauderdale, Florida USA, Nov. 1995. IEEE Computer Society.

[46] T. M. Khoshgoftaar and E. B. Allen. The impact of costs of misclassification on software quality modeling. In *Proceedings of the Fourth International Software Metrics Symposium*, pages 54–62, Albuquerque, New Mexico USA, Nov. 1997. IEEE Computer Society.

[47] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation. *Empirical Software Engineering: An International Journal*, 3(3):275–298, Sept. 1998.

[48] T. M. Khoshgoftaar and E. B. Allen. Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering*, pages 344–353, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[49] T. M. Khoshgoftaar and E. B. Allen. The stability of software quality models over multiple releases. Technical Report TR-CSE-98-25, Florida Atlantic University, Boca Raton, Florida USA, Nov. 1998.

[50] T. M. Khoshgoftaar and E. B. Allen. A comparative study of ordering and classification of fault-prone software modules. *Empirical Software Engineering: An International Journal*, 4:159–186, 1999.

[51] T. M. Khoshgoftaar and E. B. Allen. Controlling overfitting in classification-tree models of software quality. Technical report, Florida Atlantic University, Boca Raton, Florida USA, Sept. 1999.

[52] T. M. Khoshgoftaar and E. B. Allen. Logistic regression modeling of software quality. *International Journal of Reliability, Quality and Safety Engineering*, 6(4):303–317, Dec. 1999.

[53] T. M. Khoshgoftaar and E. B. Allen. Modeling the risk of software faults. Technical Report TR-CSE-00-06, Florida Atlantic University, Boca Raton, Florida USA, Feb. 2000.

[54] T. M. Khoshgoftaar and E. B. Allen. A practical classification rule for software quality models. *IEEE Transactions on Reliability*, 49(2), June 2000. In press.

[55] T. M. Khoshgoftaar, E. B. Allen, and J. C. Busboom. SMART: Software measurement analysis and reliability toolkit. Technical Report TR-CSE-98-21, Florida Atlantic University, Boca Raton, FL USA, July 1998.

[56] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 364–371, White Plains, NY, Oct. 1996. IEEE Computer Society.

[57] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, and G. P. Trio. Detection of fault-prone software modules during a spiral life cycle. In *Proceedings of the International Conference on Software Maintenance*, pages 69–76, Monterey, CA, Nov. 1996. IEEE Computer Society.

[58] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. *Computer*, 31(4):66–72, Apr. 1998.

[59] T. M. Khoshgoftaar, E. B. Allen, J. P. Hudepohl, S. J. Aud, and J. Mayrand. Selecting software metrics for a large telecommunications system. In *Proceedings of the Fourth Software Engineering Research Forum*, pages 221–229, Boca Raton, FL, Nov. 1995.

[60] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictiors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9(5):547–563, 1999.

[61] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Which software modules have faults that will be discovered by customers? *Journal of Software Maintenance: Research and Practice*, 11(1):1–18, Jan. 1999.

[62] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 6, 2000. In press.

[63] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.

[64] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. The impact of software evolution and reuse on software quality. *Empirical Software Engineering: An International Journal*, 1(1):31–44, 1996.

[65] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Predictive modeling of software quality for very large telecommunications systems. In *Proceedings of the International Communications Conference*, volume 1, pages 214–219, Dallas, TX, June 1996. IEEE Communications Society.

[66] T. M. Khoshgoftaar, B. B. Bhattacharyya, and G. D. Richardson. Predicting software errors during development using nonlinear regression models: A comparative study. *IEEE Transactions on Reliability*, 41(3):390–395, Sept. 1992.

[67] T. M. Khoshgoftaar, K. Ganesan, E. B. Allen, F. D. Ross, R. Munikoti, N. Goel, and A. Nandi. Predicting fault-prone modules with case-based reasoning. In *Proceedings of the Eighth International Symposium on Software Reliability Engineering*, pages 27–35, Albuquerque, New Mexico USA, Nov. 1997. IEEE Computer Society.

[68] T. M. Khoshgoftaar and D. L. Lanning. A neural network approach for early detection of program modules having high risk in the maintenance phase. *Journal of Systems and Software*, 29(1):85–91, Apr. 1995.

[69] T. M. Khoshgoftaar and J. C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 8(2):253–261, Feb. 1990.

[70] T. M. Khoshgoftaar, J. C. Munson, B. B. Bhattacharya, and G. D. Richardson. Predictive modeling techniques of software quality from software measures. *IEEE Transactions on Software Engineering*, 18(11):979–987, Nov. 1992.

[71] T. M. Khoshgoftaar, J. C. Munson, and D. L. Lanning. Alternative approaches for the use of metrics to order programs by complexity. *Journal of Systems and Software*, 24(3):211–221, Mar. 1994.

[72] T. M. Khoshgoftaar, A. S. Pandya, and D. L. Lanning. Application of neural networks for predicting faults. *Annals of Software Engineering*, 1:141–154, 1995.

[73] T. M. Khoshgoftaar, R. M. Szabo, and T. G. Woodcock. An empirical study of program quality during testing and maintenance. *Software Quality Journal*, 3(3):137–151, Sept. 1994.

[74] B. A. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, Jan. 1996.

[75] B. A. Kitchenham, S. L. Pfleeger, and N. E. Fenton. Towards a framework for software measurement validation. *IEEE Transactions on Software Engineering*, 21(12):929–944, Dec. 1995. See comments in [76, 86].

[76] B. A. Kitchenham, S. L. Pfleeger, and N. E. Fenton. Reply to: Comments on "Towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):189, Mar. 1997. See [75, 86, 115].

[77] D. H. Krantz, R. D. Luce, P. Suppes, and A. Tversky. *Foundations of Measurement*, volume I: Additive and Polynomial Representations. Academic Press, New York, 1971.

[78] P. A. Lachenbruch and M. R. Mickey. Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11, Feb. 1968.

[79] G. Le Gall, M. F. Adam, H. Derriennic, B. Moreau, and N. Valette. Studies on measuring software. *IEEE Journal of Selected Areas in Communications*, 8(2):234–245, Feb. 1990.

[80] D. B. Leake. CBR in context: The present and future. In D. B. Leake, editor, *Case-Based Reasoning: Experiences, Lessons, and Future Directions*, chapter 1, pages 3–30. MIT Press, Cambridge, MA USA, 1996.

[81] M. R. Lyu. Introduction. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 1, pages 3–25. McGraw-Hill, New York, 1996.

[82] J. Mayrand and F. Coallier. System acquisition based on software product assessment. In *Proceedings of the Eighteenth International Conference on Software Engineering*, pages 210–219, Berlin, Mar. 1996. IEEE Computer Society.

[83] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE–2(4):308–320, Dec. 1976.

[84] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, Dec. 1989.

[85] S. Morasca and L. C. Briand. Towards a theoretical framework for measuring software attributes. In *Proceedings of the Fourth International Symposium on Software Metrics*, pages 119–126, Albuquerque, NM USA, Nov. 1997. IEEE Computer Society.

[86] S. Morasca, L. C. Briand, V. R. Basili, E. J. Weyuker, and M. V. Zelkowitz. Comments on "Towards a framework for software measurement validation". *IEEE Transactions on Software Engineering*, 23(3):187–188, Mar. 1997. See [75, 115].

[87] J. C. Munson and T. M. Khoshgoftaar. The dimensionality of program complexity. In *Proceedings of the Eleventh International Conference on Software Engineering*, pages 245–253, Pittsburgh, Pennsylvania USA, May 1989. IEEE Computer Society.

[88] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[89] G. C. Murphy, D. N. W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology*, 7(2):158–191, Apr. 1998.

[90] J. D. Musa. Operational profiles in software reliability engineering. *IEEE Software*, 10(2):14–32, Mar. 1993.

[91] G. J. Myers. *Composite/Structured Design.* Van Nostrand Reinhold, New York, 1978.

[92] R. H. Myers. *Classical and Modern Regression with Applications.* Duxbury Series. PWS-KENT Publishing, Boston, 1990.

[93] R. J. Offen and R. Jeffery. Establishing software measurement programs. *IEEE Software*, 14(2):45–53, Mar. 1997.

[94] N. Ohlsson and H. Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22(12):886–894, Dec. 1996.

[95] N. Ohlsson, M. Helander, and C. Wohlin. Quality improvement by identification of fault-prone modules using software design metrics. In *Proceedings of the Sixth International Conference on Software Quality*, pages 2–13, Ottawa, Ontario, Canada, Oct. 1996. Sponsored by ASQC.

[96] P. Oman and S. L. Pfleeger, editors. *Applying Software Metrics.* IEEE Computer Society Press, Los Alamitos, CA, 1997.

[97] J. Pfanzagl. *Theory of Measurement.* Physica-Verlag, Wurzburg, 2d edition, 1971. In cooperation with V. Baumann and H. Huber.

[98] S. L. Pfleeger. Experimental design and analysis in software engineering. *Annals of Software Engineering*, 1:219–253, 1995.

[99] S. L. Pfleeger. Assessing measurement. *IEEE Software*, 14(2):25–26, Mar. 1997. Editor's introduction to special issue.

[100] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. A. Kitchenham. Status report on software measurement. *IEEE Software*, 14(2):33–43, Mar. 1997.

[101] G. Poels and G. Dedene. Comments on "Property-based software engineering measurement": Refining the additivity properties. *IEEE Transactions on Software Engineering*, 23(3):190–195, Mar. 1997. See [16].

[102] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

[103] F. S. Roberts. *Measurement Theory with Applications to Decisionmaking, Utility, and the Social Sciences*, volume 7 of *Encyclopedia of Mathematics and its Applications*. Addison-Wesley, Reading, Massachusetts, 1979.

[104] L. H. Rosenberg and A. Gallo. Object-oriented metrics for reliability. In *Proceedings: Fast Abstracts and Industrial Practices at the Tenth International Symposium on Software Reliability Engineering*, pages 116–132, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society. Abstract and presentation.

[105] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.

[106] N. F. Schneidewind. Software metrics validation: Space Shuttle flight software example. *Annals of Software Engineering*, 1:287–309, 1995.

[107] G. A. F. Seber. *Multivariate Observations*. John Wiley and Sons, New York, 1984.

[108] R. W. Selby and A. A. Porter. Learning from examples: Generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering*, 14(12):1743–1756, Dec. 1988.

[109] M. Shepperd and D. Ince. Design metrics and software maintainability: An experimental investigation. *Journal of Software Maintenance: Research and Practice*, 3(4):215–232, Dec. 1991.

[110] M. Stone and J. Rasp. The assessment of predictive accuracy and model overfitting: An alternative approach. *Journal of Business Finance and Accounting*, 20(1):125–131, Jan. 1993.

[111] P. F. Velleman and L. Wilkinson. Nominal, ordinal, interval, and ratio typologies are misleading. *The American Statistician*, 47(1):65–72, Feb. 1993.

[112] Verilog. *Logiscope C CodeChecker: User's Guide*, version 2.1 edition, 1996.

[113] Verilog. *Logiscope C++ CodeChecker: User's Guide*, version 2.0 edition, 1996.

[114] L. G. Votta and A. A. Porter. Experimental software engineering: A report on the state of the art. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 277–279, Seattle, WA, Apr. 1995. IEEE Computer Society.

[115] E. J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, Sept. 1988.

[116] W. E. Wong, J. R. Horgan, M. Syring, W. Zage, and D. Zage. Applying design metrics to a large-scale software system. In *Proceedings the Ninth International Symposium on Software Reliability Engineering*, pages 273–282, Paderborn, Germany, Nov. 1998. IEEE Computer Society.

[117] W. M. Zage and D. M. Zage. Evaluating design metrics on large-scale software. *IEEE Software*, 10(4):75–80, July 1993.

[118] H. Zuse. *Software Complexity: Measures and Methods*. deGuryter, Berlin, 1991.

[119] H. Zuse. Comments to the paper: Briand, Emam, Morasca: On the application of measurement theory in software engineering. *Empirical Software Engineering: An International Journal*, 2(3):313–316, 1997. See [14, 15].

[120] H. Zuse. Reply to: "Property-based software engineering measurement". *IEEE Transactions on Software Engineering*, 23(8):533, Aug. 1997. See [16].